Department of Communications and Networking

Proceedings of Seminar on Network Protocols in Operating Systems

Somaya Arianfar, Magnus Boye, Karthik Budigere, Antti Jaakkola, Aapo Kalliola, Fida Ullah Khattak, Jouni Korhonen, Arseny Kurnikov, Jonne Soininen, Nuutti Varis, Pasi Sarolahti (ed.)

Aalto University

SCIENCE + TECHNOLOGY

SEMINAR PROCEEDINGS

Aalto University publication series
SCIENCE + **TECHNOLOGY** 1/2013

Proceedings of Seminar on Network Protocols in Operating Systems

Somaya Arianfar, Magnus Boye, Karthik Budigere, Antti Jaakkola, Aapo Kalliola, Fida Ullah Khattak, Jouni Korhonen, Arseny Kurnikov, Jonne Soininen, Nuutti Varis, Pasi Sarolahti (ed.)

Aalto University
School of Electrical Engineering
Department of Communications and Networking

Aalto University publication series
SCIENCE + **TECHNOLOGY** 1/2013

ISBN 978-952-60-4997-7 (pdf)
ISSN-L 1799-4896
ISSN 1799-4896 (printed)
ISSN 1799-490X (pdf)
http://urn.fi/URN:ISBN:978-952-60-4997-7

Unigrafia Oy
Helsinki 2013

Finland

Proceedings of Seminar on Network Protocols in Operating Systems

Organized at Aalto University, Espoo, Finland on Fall 2012

Papers authored by
Somaya Arianfar
Magnus Boye
Karthik Budigere
Antti Jaakkola
Aapo Kalliola
Fida Ullah Khattak
Jouni Korhonen
Arseny Kurnikov
Jonne Soininen
Nuutti Varis

Proceedings collected together by Pasi Sarolahti

Table of Contents

Foreword

The Linux networking stack tends to evolve fairly rapidly, and while there are some excellent documentation written in the past, most of the past documentation has gotten (at least partially) outdated over time. The seminar on Network Protocols in Operating Systems was arranged in Aalto University, Department of Communications and Networking to gain a better understanding of the current status of the networking implementation in the Linux kernel. The seminar had 10 participants in addition to the undersigned as the seminar organizer. Each participant was assigned a module from the Linux networking implementation, on which a short paper was to be written. These proceedings contain the final output of this work.

Our group had meetings every two weeks between September 2012 and December 2012 to follow the progress of work, and to discuss different ancillary issues related to the seminar topic, such as techniques for paper writing and reviewing, or tools for Linux kernel debugging.

The seminar simulated a small scientific workshop: draft versions of papers were returned by mid-November, and peer-reviewed anonymously by 3 co-participants of the seminar. Based on the review feedback final versions of papers were prepared to be included in the proceedings. During a public final workshop the participants gave 20-minute presentations of their topics. About 20 people attended the final workshop, and actively participated discussion.

For myself this seminar was a useful and fun experience in getting more familiar with various aspects in Linux kernel protocol implementations. I believe that the papers included in these proceedings will be useful material also for many others interested in how the Linux networking stack is implemented.

Pasi Sarolahti
Seminar organizer

Linux Kernel Application Interface

Arseny Kurnikov
Aalto University School of Electrical Engineering
PO Box 13000, FI-00076 Aalto
Espoo, Finland
arseny.kurnikov@aalto.fi

ABSTRACT

This paper describes different interfaces exposed by Linux kernel to applications. The focus is on the socket application interface that provides support for BSD sockets as defined in *Berkeley Software Architecture Manual, 4.4*. Other topics cover `/proc/sys` file system and system control (*sysctl*), input/output control *ioctl* mechanism and *Netlink* sockets.

Different application programming interfaces (APIs) are used for different purposes though their functionality overlaps sometimes. The means of extending APIs and introducing new features vary between mechanisms. To build a reliable user space application and to keep the kernel stable and unpolluted it is essential to know the details of the approaches and to choose a suitable API whenever needed. The API mechanisms described in this paper are compared from the point of view of functionality and extendability. The internal API details are described.

Keywords

Kernel API, Linux sockets

1. INTRODUCTION

A kernel runs in the protected mode of CPU. It is not possible for user space applications to access kernel data structures and functions directly. Thus a kernel provides its services to the applications through the means of Application Programming Interface. API is needed to allow user space applications to interact with the kernel.

API is a set of functions and protocols. It describes the rules how to pass arguments and messages to request a specific kernel service. These rules remain the same throughout the kernel development process to maintain backwards compatibility.

The Linux socket application programming interface is implemented according to 4.4BSD specification [13]. API documentation is available in the form of manual pages [10]. Examples of typical socket usage are available as unoficial tutorials, i.e. [3]. The focus of this work is what happens inside the kernel when API functions are invoked by a process. An overall picture of all kernel interfaces is given in [2].

Another topic discussed is Netlink sockets. The formal specification of the Netlink protocol is available as RFC 3549 [11]. The Netlink manual page [7] describes the basic operation of the Netlink protocol and message structures. Netlink is responsible for transporting data between communication ends, and one of the protocols on top of it is Routing Netlink protocol [8].

Ioctl and sysctl are two mechanisms to tune different parameters. Ioctl controls devices [6], sysctl deals with kernel parameters [5]. Ioctl is used for changing socket parameters, too, but it is outdated and Netlink was introduced to substitute ioctl commands. Sysctl exposes networking parameters as well. They are on a module scale, as opposite to per-device ioctl commands. The list of sysctl IPv4 networking parameters is available as part of the Linux source code under *Documentation/ip-sysctl.txt*. Finally, for setting/getting options (flags) of a single socket two functions from the socket API are available: `setsockopt` and `getsockopt` [9].

The focus of this work is on the socket API. In the next chapter, an overview of the API logic is given. In chapter 3, more details of some operations are described. Chapter 4 discusses the means of extending APIs and compares different mechanisms from different points of views. The last chapter summarizes the information and gives the conclusions.

2. OVERVIEW

This chapter describes several API mechanisms in general. The first section presents socket API functions and structures. The next two sections are dedicated to ioctl and sysctl. In the last section, Netlink sockets are introduced.

2.1 Socket API

A socket is an abstraction that represents one end of a communicating channel between two processes. These processes may be on different machines in a network, or on the same computer. One of the processes might be the kernel itself.

A socket belongs to a protocol family and works according

Table 1: Socket API functions

Function	Description
socket	create a new socket
bind	bind a socket to an address
connect	connect to another end
accept	accept a new connection
socketpair	create a pair of connected sockets
poll	check for some activity on a socket
ioctl	run ioctl command on a socket
listen	wait for incoming connections
sendmsg	send a message
recvmsg	receive a message
setsockopt	set socket options
getsockopt	get socket options
shutdown	block sending or receiving
close	close the socket

to some protocol. A protocol family is a collection of protocols. For example, IPv4 protocol suite contains TCP, UDP and IP protocols and the part of the Internet Control Message Protocol (ICMP) related to "ping" messages. Other subsystems, like the Internet Group Management Protocol, are also parts of the IPv4 protocol family but they do not provide the functions to manipulate the sockets of their type.

Data can be read from or written to the socket. Other socket operations include waiting for a new connection, binding a socket to an address, accepting a new connection, setting and getting socket options. Socket API functions are given in Table 1 with a short description.

The socket functions are exposed to the applications through the use of system calls. System calls is the main mechanism used by all processes to interact with the kernel. During the system call the execution mode is switched from user space to kernel space. Then the kernel performs the required service and returns the results to the process.

Each system call has an assigned system call number. Socket system calls are multiplexed through one system call number *socketcall*. A subcode specifies which socket function to invoke. There is also a non-multiplexed family for socket system calls.

The structure that represents a socket in the kernel is `struct socket`. It contains the fields for a socket state, a socket type, socket flags, protocol specific socket operations, a corresponding file descriptor, underlying `struct sock`. The latest structure is the internal protocol independent socket representation. It includes the pointers to protocol specific operations provided by the protocol family.

Figure 1 represents the relationships of the socket related structures. Operations structures contain function pointers for socket operations. They differ depending on how generic the implementation is. Protocol level functions implement protocol specific features. Protocol family callbacks perform some common actions for a set of protocols and usually call the protocol-specific handlers after that. The most generic are file operations.

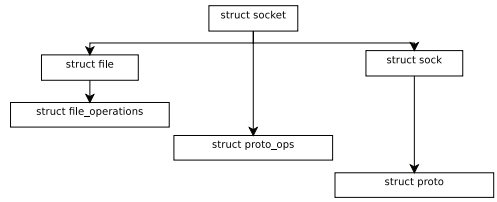

Figure 1: Socket data structures

Socket can be referred by the file descriptor it is associated with. This is the reason why the socket structure contains a file descriptor. Sending and receiving a message can be done through the file operations `read()` and `write()`, too. They are described in more details in the next chapter.

2.2 Ioctl mechanism

Ioctl is an acronym for input output control. The purpose of ioctl is to setup different parameters of the devices or perform device specific actions. It can be used with sockets, too. The function `ioctl()` takes three arguments: the socket descriptor, the ioctl command and the command argument.

A protocol family has a set of the ioctl commands. Some of the commands go on to the socket level and are handled by the protocol implementation. For example, IPv4 protocol family contains operations for manipulating routing tables, ARP tables, device configuration. UDP protocol implements two commands: `SIOCOUTQ` and `SIOCINQ` to report the length of the outgoing and incoming queues.

Ioctls lack well-defined interface and structure [1]. There is no agreement on what the command argument can be, so it is up to a programmer to decide if it is a pointer to a complicated data structure or an index in some internal table and so on. The commands vary between devices and new commands can be easily introduced though it is preferred to have unique numbers for the commands and the conflicts are very often. Ioctls cannot be transported over the network, because the argument might be the pointer to a structure in the user-space. That is why ioctl is deprecated but is still used in legacy systems. To solve the ioctl problems Netlink sockets were introduced.

2.3 Netlink sockets

A Netlink socket is a socket implementing kernel-user communicating protocol [7]. Netlink API is available for user space processes and kernel modules. It is an address family and follows the common scheme for address families: it registers the protocol family operation for creating a socket, in the creating function it allocates the socket structure and sets up the socket operations.

The structure `struct netlink_sock` contains the fields for underlying `struct sock`, port identificator that serves the role of the netlink socket address, flags, multicast groups, socket state. If a process owns only one Netlink socket it sets the port identificator to its process `id`. The kernel can also do it automatically. For several Netlink sockets it is the

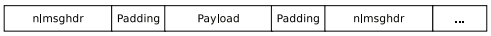

Figure 2: Netlink message

responsibility of the process to keep the addresses unique. The kernel Netlink socket address is always 0.

Netlink sockets are of type `SOCK_RAW` or `SOCK_DGRAM`. Applications fill up the header structure `struct nlmsghdr`. It contains the fields for the length of the message, the type of the content, flags, sequence number and the port of the sending process. The payload follows each header [7]. The Netlink message format is given on Figure 2.

One of the protocol using Netlink sockets is the Routing Netlink protocol. It is described in the next chapter as a use case: modifying a routing table.

2.4 Sysctl mechanism

Sysctl is a mechanism to tune different kernel parameters. It represents an hierarchy under */proc/sys* directory. Directories in this hierarchy are different sysctl subsystems, like *fs* for filesystem, *net* for networking, *kernel* for core kernel parameters and so on. Each leaf in this hierarchy is a file that contains a value for one parameter. Since it is a file, the generic mechanism of using `struct file_operations` to define functions pointers for reading and writing operations is utilized.

Several subsystems register themselves under sysctl directory for networking. They include IPv4 and IPv6 protocol families, unix sockets, netfilter and core subsystem. The registration function is `register_net_sysctl()` that in turn calls `__register_sysctl_table()`. The main argument to this function is a pointer to a node of the hierarchy.

The structure that represents a node in the sysctl hierarchy is `struct ctl_table`. It contains a name for */proc/sys* entry, a pointer to the data that is exposed through this entry, an access mode and a callback handler for text formatting. The callback handles the data passed to read and write functions. There are several default handlers for strings, vectors of integers and vectors of long integers. The default handlers perform necessary formatting and conversions depending on the type of parameter.

The implementation of the sysctl backend resides under the implementation of */proc* filesystem. It registers and unregisters sysctl nodes, looks up the entries, provides callbacks for read and write operations. The */proc* filesystem not only contains configurable parameters under sysctl subsystem but also provides statistics about the kernel and processes running in the system. For networking the statistics is available under the */proc/net* directory. It includes packet and socket counters, detailed information about protocols used, the routes registered, the network devices available and so on.

Modules expose configuration parameters through sysctl. For example, TCP protocol allows to configure the congestion control mechanism, keepalive interval, keepalive time and many others. IP has the parameters for default TTL, whether or not IP forwarding is enabled, the range of the local ports and so on.

3. OPERATIONS AND DETAILS

In this chapter, some of the operations are discussed in more details. In particular, it is shown what happens when a process invokes a system call. The following sections tells about creating a socket and sending a message in more details. Finally, adding a route with the Routing Netlink protocol is depicted.

3.1 Kernel and processes boundary

From the application side the entering point for invoking kernel system calls is the C library. To make a system call the original convention on Linux/i386 was to use interrupt *0x80*. The modern approach is by `sysenter/sysexit` instructions. The C library defines the macro `ENTER_KERNEL` that provides an appropriate way to switch between CPU modes. The system call number should be put on *EAX* register.

The C library functions for socket operations are declared in *sys/socket.h* and implemented in assembler files depending on architecture. The library includes stub versions of the functions that are not supported by the architecture. A stub function sets the error number to `ENOSYS` and returns -1.

Implementations of all socket library functions are similar and consist of the following steps:

- save registers
- put a subcode socket function number in *EBX*
- put the rest of the arguments address in *ECX*
- do `ENTER_KERNEL`
- restore registers
- return the result of the system call or set the error number

Not only the execution mode is separated for the kernel and the applications. Processes cannot access the kernel memory and data, too. There are functions to copy data from user space to kernel space and backwards: `copy_from_user()` and `copy_in_user()`. Their implementation depends on the architecture. For simple values, like `char` and `int`, there are functions: `put_user()` and `get_user()`.

3.2 Creating a socket

Figure 3 shows the procedure of creating a new socket. An application calls `socket()` library function that invokes the `socket()` system call. It performs the following steps:

- Check that arguments are valid.
- Allocate a new socket and inode and bind them together.
- Call `create()` provided by a protocol family.

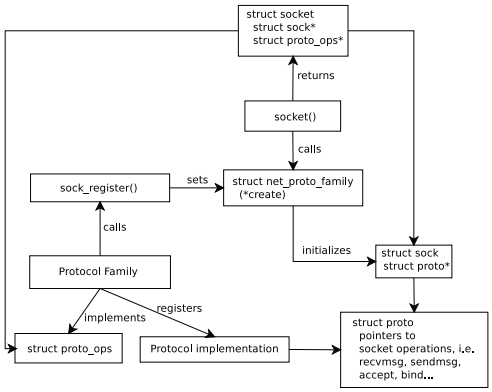

Figure 3: New socket

- Allocate a new file descriptor and map it to the socket.

The file descriptor is saved in the `file` field of the socket structure. In the `private_data` of the file descriptor structure there is a pointer to the socket that is mapped to this file descriptor. Thus given a socket, the corresponding file descriptor can be retrieved and vice versa.

Protocol families register themselves within the socket system by calling `sock_register()`. The argument of this function is a pointer to `struct net_proto_family`. This structure describes a protocol family and contains a function pointer to create a socket of a certain type. The creating function is responsible for allocating `struct sock`, setting up the operations field of `struct socket` and calling a protocol specific initialization function.

The aforementioned operations field of the socket structure serves the role of setting callbacks for the actions performed on sockets. It represents `struct proto_ops` and works with `struct socket`. The implementations of message sending and receiving callbacks in `struct proto_ops` get the corresponding `struct sock` and call the appropriate function from the protocol implementation. On the other hand, functions like `bind()` and `ioctl()` are mostly implemented by the protocol family as they are common for all protocols in the family. Though for example, RAW sockets have their own `bind()` implementation by the protocol.

The structure that describes each protocol is `struct proto`. The protocol registration function is `proto_register()`. It adds the protocol to the list maintained in *net/core/sock.c*. Protocol families register their protocols at the initialization. The fields of `struct proto` include the function pointers for protocol specific socket initialization, connecting, disconnecting, setting socket options and sending/receiving messages.

3.3 Sending messages

There are several ways to send and receive a message.

A socket is a file. Thus `write()` can be used to send a message. The corresponding file descriptor contains a pointer to `struct file_operations`. The socket write function does necessary conversions of the arguments so it is possible to call `sock_sendmsg()`. This function either uses the security method for sending a message or proceeds to the operation saved in the `ops` field of the socket structure. Since `write()` is a generic function it is not possible to set socket specific flags.

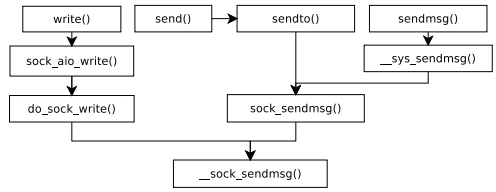

Figure 4: Sending a message

Another option is `sendto()` system call. It takes the following arguments:

- `int fd` – the file descriptor
- `void* buff` – the buffer with the message
- `size_t len` – the length of the message
- `unsigned int flags` – sending flags
- `struct sockaddr* addr` – the destination address
- `int addr_len` – the length of the address

It looks up the socket associated with the given file descriptor, fills up the structure `struct msghdr`, moves the address to kernel space and calls `sock_sendmsg()`.

The system call `send()` is implemented through `sendto()` by setting the destination address and its length to zero.

Finally, there is `sendmsg()` system call. Its arguments are: file descriptor, `struct msghdr` that contains the message, sending flags. It gives the most control to the application, because the message header structure is the one that is used internally. It contains `struct iovec` that provides scattering capabilities. So the data can be put to this array from different sources and sent as one message.

Figure 4 shows the sending functions and their common point at `__sock_sendmsg`.

The receving functions are: `read()` that is a generic one for files, `recvfrom()` that allows to specify flags, `recv()` that is implemented through `recvfrom()`, and `recvmsg()`. So the receiving message framework is a mirror of the sending one.

3.4 Adding a route

One of the protocols on top of netlink sockets is the routing netlink protocol `NETLINK_ROUTE`. The messages of this protocol include routing tables modification and retrieving routing information, controlling neighbor lists and links available. The module *rtnetlink* creates a kernel netlink socket

and allows other modules to register message types, so that different protocol families register routing manipulation messages of their own.

`rtnl_register()` is the registration function. Its arguments are: a protocol family that implements the callback for the message, the message type, the pointers to the callbacks. There are three functions that can be registered for any message type: `doit()`, `dumpit()` and `calcit()`. `doit()` is used when a message performs some action, like adding a route. `dumpit()` is implemented when a message is a request for some information. `calcit()` is a helper function for calculating the buffer size for the information requested.

Message receiving function of *rtnetlink* gets the kind of the message: request for action or request for information. If the flags passed through `struct nlmsghdr` contain `NLM_F_DUMP` then the message is a request for information and the dump process is started. If `calcit` is available then it calculates the minimum buffer size for the reply before starting the dump.

If the message is a request for action and a user has sufficient capabilities then *rtnetlink* gets the `doit()` callback associated with the message type, extracts attributes from the message and calls the callback passing in the attributes. The attributes allow each message to contain subsequent data.

A set of messages for manipulating routing tables in IPv4 include: `RTM_NEWROUTE`, `RTM_DELROUTE` and `RTM_GETROUTE` [4]. The first two register `doit()` callbacks and the third one registers the `dumpit()` function.

The structure that represents a payload for the routing message is `struct rtmsg`. It contains the following fields:

- `rtm_family` – the protocol family of the route, i.e. IPv4 or IPv6 separately registers the messages for their routing tables;
- `rtm_dst_len` – the destination address length;
- `rtm_src_len` – the source address length;
- `rtm_tos` – the type of service;
- `rtm_table` – the routing table to operate on;
- `rtm_protocol` – the type of the route, i.e. static or dynamic;
- `rtm_scope` – is it a global route or the one internal for a protocol family;
- `rtm_type` – the type of the route, i.e. unicast, broadcast, local, multicast, unreachable;
- `rtm_flags` – the flags.

For IPv4 the handling of the messages is done in the Forwarding Information Database (FIB) implementation, where the registration of messages takes place. The processing function for a new route is `inet_rtm_newroute()` that parses the message to the format that FIB understands, looks up the routing table and creates a new routing entry.

In order for a user-space application to add a new route the procedure is as follows. A routing message structure `struct rtmsg` is filled in with the route details. The netlink header structure is initialized with the message type `RTM_NEWROUTE`, the destination port 0 and the protocol family that the route belongs to. Then the Netlink socket is created and the message with the header structure and `rtmsg` as a payload is sent. If the user has enough permissions then the route will be added.

4. EXTENDING API

APIs are different from the point of view of functionality, and extendability. This chapter discusses what is needed to extend an API.

In [12] the process of creating Linux kernel modules is described. The kernel has modular structure so that it is easier to introduce new features. Modules can be loaded and unloaded without rebooting the kernel.

The most rich API is system calls. They provide the way to request a kernel to perform some action. A system call cannot be implemented in a module because the table that contains system call functions is static. The kernel initializes the table during the booting and never changes it after that. The table is not exported to the modules so that it cannot be modified.

To add a system call the following steps are required. First, the syscall entry is added to `entry.S`. This file is architecture specific. For *x86* processors the table is defined in `syscall_xx.tbl` where `xx` is 32 bits or 64. Then the actual system call is implemented. Finally, `unistd.h` is modified to include the new system call and to change the overall number of them in the kernel.

Adding a new ioctl handler does not give as much flexibility as adding a new system call. Ioctl commands are bound to the devices. To add an ioctl handler it is necessary to fill up the structure `struct file_operations` and register a device with this structure. Another possibility is to implement a protocol family so that `socket_ioctl` system call ends up in the handler. Each handler has two parameters: the ioctl command and the command parameter.

As was already mentioned introducing new ioctls should be done with care. The instructions to add a new ioctl are available in *Documentation/ioctl/ioctl-number.txt*. It is not restricted what the parameter can be. So if it is a pointer to a structure then ioctl are flexible.

Another way to allow a module to expose API is to create a Netlink socket and communicate with processes via Netlink messages. In practice, it means designing and implementing a new protocol on top of Netlink.

Finally, the natural way to expose module parameters is sysctl. Only one step is needed for it: to register a new `ctl_table`. This is done by filling up the control table structure with the file name in the hierarchy, the pointer to the

Table 2: API comparison

Method	Functionality	Extendability
System call	Very rich	Not in a module
Netlink	Rich	New protocol
Sysctl	For module parameters	New ctl_table
Ioctl	For device control	Limited

data that will be exposed through the file, the access mode and the handler for text formatting. Then `register_sysctl` should be called specifying the path to the directory where the file is to be put. The filled in table is passed as a second parameter to the function. Thus the modification of the file will result in calling the handler. And if one of the default handlers is used then the modification of the file will result in modification of the data pointed by the field in `struct ctl_table`.

Table 2 lists APIs described in this paper and compares them from the point of view of functionality and extendability.

5. CONCLUSIONS

In this work, different Linux kernel application interfaces are described. They are used for different purposes. They are also different from the point of view of documentation and ease of extending and introducing new features.

Systems calls that socket API is based on, is the main way the processes interact with the kernel. Implementing a new system call requires modification of the kernel and is architecture specific.

Ioctl is used to control devices. This framework is legacy and was a subject to discussion for re-design. To overcome its limitations Netlink sockets were introduced.

Netlink sockets are based on a well-known sockets paradigm. There is a lot of documentation available. Introducing new features means implementing a new protocol. It requires designing structures of messages and rules for exchanging them. The functionality is close to that of system calls.

Finally, sysctl system is used to expose different kernel parameters as files. Thus, the changing of the parameter requires writing a new value to the file. And the file can be read to know the current value of the parameter.

Extending APIs should be done with clever thought of the system design. Some of the frameworks put constraints on what can be done, the others are more flexible but at the same time they are more error-prone. The choice of the API depends on the nature of tasks that should be performed.

6. REFERENCES

[1] Ioctl discussion. `http://yarchive.net/comp/linux/ioctl.htm`.

[2] Linux kernel map. `http://www.makelinux.net/kernel_map/`.

[3] *Sockets Tutorial*. `http://www.linuxhowtos.org/C_C++/socket.htm`.

[4] N. Horman. *Understanding And Programming With Netlink Sockets*, 0.3 edition, 2004.

[5] Linux man-pages project. *sysctl(2)*, 3.27 edition, 1999.

[6] Linux man-pages project. *ioctl(2)*, 3.27 edition, 2000.

[7] Linux man-pages project. *netlink(7)*, 3.27 edition, 2008.

[8] Linux man-pages project. *rtnetlink(7)*, 3.27 edition, 2008.

[9] Linux man-pages project. *getsockopt(2)*, 3.27 edition, 2009.

[10] Linux man-pages project. *socket(2)*, 3.27 edition, 2009.

[11] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. *Linux Netlink as an IP Services Protocol, RFC 3549*. IETF, Network Working Group, 2003.

[12] P. J. Salzman. *The Linux Kernel Module Programming Guide*, 2.6.4 edition, 2007.

[13] University of California, Berkeley, CA 94720. *Berkeley Software Architecture Manual*, 4.4bsd edition.

Implementation of Transmission Control Protocol in Linux

Antti Jaakkola
Aalto University
Department of Communication and Networking
antti.t.jaakkola@aalto.fi

ABSTRACT

Transmission Control Protocol is the most used transmission layer protocol in the Internet. In addition to reliable and good performance in transmission between two nodes, it provides congestion control mechanism that is a major reason why Internet has not collapsed. Because of its complicated nature, implementations of it can be challenging to understand. This paper describes fundamental details of Transmission Control Protocol implementation in Linux kernel. Focus is on clarifying data structures and segments route through TCP stack.

1. INTRODUCTION

In May 1974 Vint Cerf and Bob Kahn published paper where they described an inter-networked protocol, which central control component was Transmission Control Program [3, 2].Later it was divided into modular architecture and in 1981 Transmission Control Protocol (TCP), as it is know today, was specified in RFC 793 [8].

Today, TCP is the most used transmission layer protocol in the Internet [4] providing reliable transmission between two hosts through networks [8]. In order to gain good performance for communication, implementations of TCP must be highly optimized. Therefore, TCP is one of the most complicated components in Linux networking stack. In kernel 3.5.4, it consists of over 21000 lines of code under net/ipv4/ -directory (all tcp*.c files together), while IPv4 consist of less than 13000 lines of code (all ip*.c files in the same directory). This paper explains the most fundamental data structures and operations used in Linux to implement TCP.

TCP provides reliable communication over unreliable network by using acknowledgment messages. In addition to provide resending of the data, TCP also controls its sending rate by using so-called 'windows' to inform the other end how much of data receiver is ready to accept.

As parts of the TCP code are dependent on network layer implementation, the scope of this paper is limited to IPv4 implementation as it is currently supported and used more widely than IPv6. However, most of the code is shared between IPv4 and IPv6, and tcp_ipv6.c is the only file related to TCP under net/ipv6/. In addition, TCP congestion control will be handled in a separate paper, so it will be handled very briefly. If other assumptions is made it is mentioned in the beginning of the related section.

Paper structure will be following: First section "Overview of implementation" will cover most important files and basic data structures used by TCP (**tcp_sock**, **sk_buff**), how data is stored inside these structures and how different queues are implemented, what timers TCP is using and how TCP sockets are kept in memory. Then a socket initialization and data flows through TCP are discussed. Section "Algorithms and optimizations" will handle logic of TCP state machine, briefly look into congeston control and explain what is TCP fast path.

2. OVERVIEW OF IMPLEMENTATION

In this section basic operation of TCP in Linux will be explained. It covers the most fundamental files and data structures used by TCP, as well as functions used when we are sending to or receiving from network.

The most important source files of implementation are listed in the Table 1. In addition to net/ipv4/, where most TCP files are located, there are also few headers located in include/net/ and include/linux/ -directories. [1]

Table 1: Most important files of TCP

File	Description
tcp.c	Layer between user and kernel space
tcp_output.c	TCP output engine. Handles outgoing data and passes it to network layer
tcp_input.c	TCP input engine. Handles incoming segments.
tcp_timer.c	TCP timer handling
tcp_ipv4.c	IPv4 related functions, receives segments from network layer
tcp_cong.c	Congestion control handler, includes also TCP Reno implementation
tcp_[veno|vegas|..].c	Congestion control algorithms, named as tcp_NAME.c
tcp.h	Main header files of TCP. **struct tcp_sock** is defined here. Note that there is tcp.h in both include/net/ and include/linux/

[1]Note that this paper is based on kernel version 3.5.3. In Linux 3.7, the new UAPI header file split moved some header files to new locations.

2.1 Data structures

Data structures are crucial sections of any software in order of performance and re-usability. As TCP is a highly optimized and remarkably complex entirety, robust understanding of data structures used is mandatory for mastering the implementation.

2.1.1 struct tcp_sock

struct tcp_sock (include/linux/tcp.h) is the core structure of TCP. It contains all the information and packet buffers for certain TCP connection. Figure 1 visualizes how this structure is implemented in Linux. Inside **tcp_sock** there is a few other, more general type of sockets. As a next, more general type of socket is always first member of socket type, can a pointer to socket be type-casted to other type of socket. This allows us to make general functions that handles with, for example, **struct sock**, even in reality pointer would be also a valid pointer to **struct tcp_sock**. Also depending on the type of the socket can different structure be as a first member of the certain socket. For example, as UDP is connection-less protocol, first member of **struct udp_sock** is **struct inet_sock**, but for **struct tcp_sock** first member must be **struct inet_connection_sock**, as it provides us features needed with connection-oriented protocols.

From Figure 1 it can be seen that TCP has many packet queues. There is receive queue, backlog queue and write queue (not in figure) under **struct sock**, and pre-queue and out-of-order queue under **struct tcp_sock**. These different queues and their functions are explained in detail in section 2.1.2.

struct inet_connection_sock (include/net/ inet_connection_sock) is a socket type one level down from the **tcp_sock**. It contains information about protocol congestion state, protocol timers and the accept queue.

Next type of socket is **struct inet_sock** (include/net/inet_sock.h). It has information about connection ports and IP-addresses.

Finally there is general socket type **struct sock**. It contains two of TCP's three receive queues, sk_receive_queue and sk_backlog, and also queue for sent data, used with retransmission.

2.1.2 Data queues

Incoming data queues are used as data storage before user reads the data to user space. These queues are implemented as double linked ring-list of **struct sk_buff**s (see section 2.1.3).

When user reads data from the socket, socket will be marked as being in use to avoid conflicts. However, incoming segments must be saved even when the socket is in use. Therefore, socket has several queues for incoming data: receive queue, pre-queue, and backlog queue. In addition to these, out-of-order queue is used as temporary storage for segments arriving out of order.

In the normal case when segment arrives and user is not waiting for the data, segment is processed immediately and the data is copied to the receive queue. Data will be copied to user's buffer when application reads data from the socket.

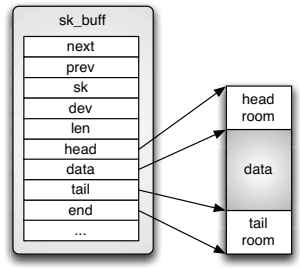

Figure 2: Data storage inside structure sk_buff

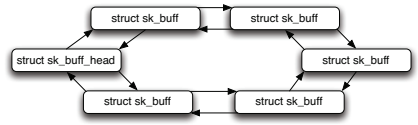

Figure 3: Ring-list of sk_buffs

If user is using blocking IO and the receive queue does not have as many bytes as requested, will the socket be marked as waiting for data. When the new segment arrives, it will be put to pre-queue and waiting process will be awakened. Then the data will be handled and copied to user's buffer.

If user is handling segments (socket is marked as being in use) at the same time when we receive a new one, it will be put to the backlog queue, and user context will handle the segment after it has handled all earlier segments from other queues.

In addition to incoming data queues, there is also outgoing data queue known as write queue. It is implemented and used in the same way as incoming buffers. Segments will be put to write queue as user writes data to socket, and they will be removed when an acknowledgment arrives from the receiver.

Figure 4 visualizes use of receive, pre- and backlog-queues.

2.1.3 struct sk_buff

struct sk_buff (located in include/linux/skbuff.h) is used widely in the network implementation in Linux kernel. It is a socket buffer containing one slice of the data we are sending or receiving. In Figure 2 we see how data is stored inside structure. Data is hold in the continuous memory area surrounded by empty spaces, head and tail rooms. By having these empty spaces more data can be added to before or after current data without needing to copy or move it, and minimize risk of need to allocate more memory. However, if the data does not fit to space allocated, it will be fragmented to smaller segments and saved inside **struct skb_shared_**

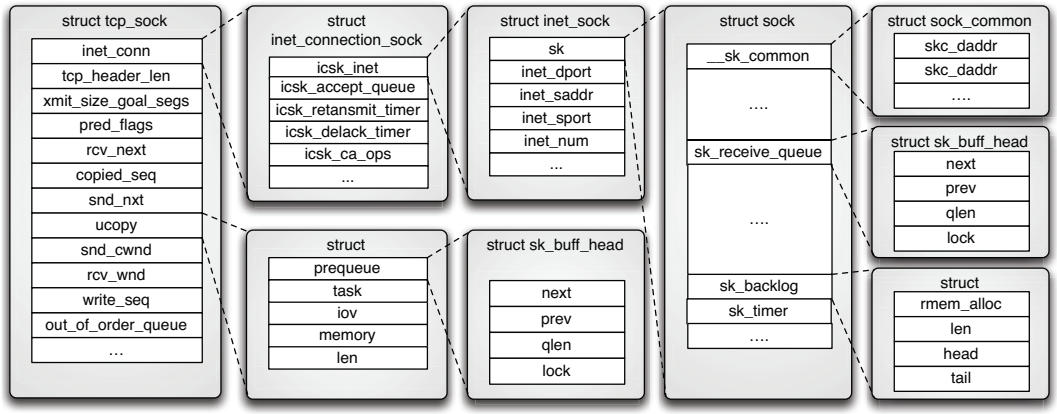

Figure 1: Socket structures involved in TCP connection

info that lives at the end of data (at the end pointer).

All the data cannot be held in one large segment in the memory, and therefore we must have several socket buffers to be able to handle major amounts of data and to resend data segment that was lost during transmission to receiver. Because of that need of network data queues is obvious. In Linux these queues are implemented as double linked ring-lists of sk_buff structures (Figure 3). Each socket buffer has a pointer to the previous and next buffers. There is also special data structure to represent the whole list, known as **struct sk_buff_head**, that is used to indicate the first and the last members of ring list. More detailed information about the data queues is in section 2.1.2.

In addition data pointers, sk_buff also has pointer to owning socket, device from where data is arriving from or leaving by and several other members. All the members are documented in skbuff.h.

2.1.4 Hash tables

Hash table is a data structure that is used to map a given key to corresponding value.

Sockets are located in kernel's hash table from where them are fetched when a new segment arrives or socket is otherwise needed. Main hash structure is **struct inet_hashinfo** (include/net/inet_hashtables.h), and TCP uses it as a type of global variable tcp_hashinfo located in net/ipv4/tcp_ipv4.c.

struct inet_hashinfo has three main hash tables: One for sockets with full identity, one for bindings and one for listening sockets. In addition to that, full identity hash table is divided in to two parts: sockets in TIME_WAIT state and others.

As hash tables are more general and not only TCP specific part of kernel, this paper will not go into logic behind these more deeply.

2.1.5 Other data structures and features

There are also several other data structures that must be known in order to understand how TCP stack works. **struct proto** (include/net/sock.h) is a general structure presenting transmission layer to socket layer. It contains function pointers that are set to TCP specific functions in net/ipv4/tcp_ipv4.c, and applications function calls are eventually, through other layers, mapped to these.

struct tcp_info is used to pass information about socket state to user. Structure will be filled in function tcp_get_info(). It contains values for connection state (Listen, Established, etc), congestion control state (Open, Disorder, CWR, Recovery, Lost), receiver and sender MSS, rtt and various counters.

To provide reliable communication with good performance, TCP uses four timers: Retransmit timer, delayed ack timer, keep-alive timer and zero window prope timer. Retransmit, delayed ack and zero window probe timers are located in **struct inet_connection_sock**, and keep-alive timer can be found from **struct sock** (Figure 1).

Although there is dedicated timer handling file net/ipv4/tcp_timer.c, timers are set and reset in several locations in the code as a result of events that occur.

2.2 Socket initialization

TCP functions available to socket layer are set to previously explained (section 2.1.5) **struct proto** in tcp_ipv4.c. This structure will be held in **struct inet_protosw** in af_inet.c, from where it will be fetched and set to sk->sk_prot when user does socket() call. During socket creation in the function inet_create() function sk->sk_prot->init() will be called, which points to tcp_v4_init_sock(). From there the real initialization function tcp_init_sock() will be called.

Address-family independent initialization of TCP socket occurs in tcp_init_sock() (net/ipv4/tcp.c). The function will

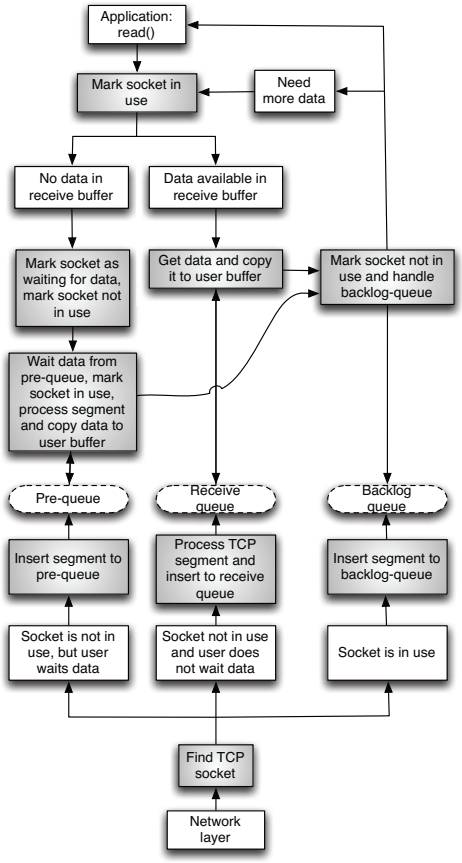

Figure 4: Use of different incoming data queues (without out of order queue)

be called when socket is created with socket() system call. In that function fields of structure tcp_sock are initialized to default values. Also out of order queue will be initialized with skb_queue_head_init(), pre-queue with tcp_prequeue_init(), and TCP timers with tcp_init_xmit_timers(). At this point, state of the socket is set to TCP_CLOSE.

2.2.1 Connection socket

Next step to do when user wants to create a new TCP connection to other host is to call connect(). In the case of TCP, it maps to function inet_stream_connect(), from where sk->sk_prot->connect() is called. It maps to TCP function tcp_v4_connect().

tcp_v4_connect() validates end host address by using ip_route_connect() function. After that inet_hash_connect() will be called. inet_hash_connect() selects source port for our socket, if not set, and adds the socket to hash tables. If everything is fine, initial sequence number will be fetched from secure_tcp_sequence_number() and the socket is passed to tcp_connect().

tcp_connect() calls first tcp_connect_init(), that will initialize parameters used with TCP connection, such as maximum segment size (MSS) and TCP window size. After that tcp_connect() will reserve memory for socket buffer, add buffer to sockets write queue and passes buffer to function tcp_transmit_skb(), that builds TCP headers and passes data to network layer. Before returning tcp_connect() will start retransmission timer for the SYN packet. When SYN-ACK packet is received, state of socket is modified to ESTABLISHED, ACK is sent and communication between nodes may begin.

2.2.2 Listening socket

Creation of listening socket should be done in two phases. Firstly, bind() must be called to pick up port what will be listened to, and secondly, listen() must be called.

bind() maps to inet_bind(). Function validates port number and socket, and then tries to bind the wanted port. If everything goes fine function returns 0, otherwise error code indicating problem will be returned.

Function call listen() will become to function inet_listen(). inet_listen() performs a few sanity checks, and then calls function inet_csk_listen_start(), which allocates memory for socket accept queue, sets socket state to TCP_LISTEN and adds socket to TCP hash table to wait incoming connections.

2.3 Data flow through TCP in kernel

Knowing the rough route of incoming and outgoing segments through the layer is on of the most important part of TCP implementation to understand. In this section a roughly picture of it in most common cases will be given. Handling of all the cases is not appropriate and possible under the limits of this paper.

In this section it is assumed that DMA (CONFIG_NET_DMA) is not in use. It would be used to offload copying of data to dedicated hardware, thus saving CPU time. [1]

2.3.1 From the network

Figure 5 shows us a simplified summary about incoming data flow through TCP in Linux kernel.

In the case of IPv4, TCP receives incoming data from network layer in tcp_v4_rcv() (net/ipv4/tcp_ipv4.c). The function checks if packet is meant for us and finds the matching TCP socket from the hash table using IPs and ports as the keys. If the socket is not owned by user (user context is not handling the data), we first try to put the packet to pre-queue. Pre-queuing is possible only when user context is waiting for the data. If pre-queuing was not possible, we pass the data to tcp_v4_do_rcv(). There socket state is checked. If state is TCP_ESTABLISHED, data is passed to tcp_rcv_established(), and copied to receive queue. Otherwise buffer is passed to tcp_rcv_state_process(), where all the other states will be handled.

If the socket was not owned by user in function tcp_v4_rcv(), data will be copied to the backlog queue of the socket.

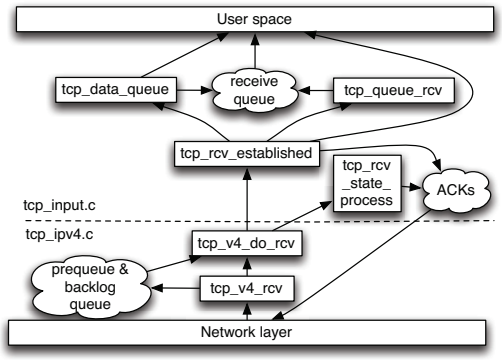

Figure 5: Data flow from network to user

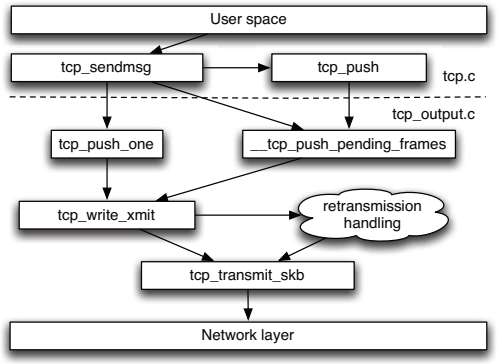

Figure 6: Data flow from user to network

When user tries to read data from the socket (tcp_recvmsg()), queues must be processed in order. First receive queue, then data from pre-queue will be waited, and when the process ready to release socket, packets from backlog will be copied to the receive queue. Handling of the queues must be preserved in order to ensure that data will be copied to user buffer in the same order as it was sent.

Figure 4 visualizes the overall queuing process.

2.3.2 From the user

Figure 6 shows us a simplified summary about outgoing data flow through TCP in Linux kernel.

When user-level application writes data to TCP socket, first function that will be called is tcp_sendmsg(). It calculates size goal for segments and then creates sk_buff buffers of calculated size from the data, pushes buffers to write queue and notifies TCP output engine of new segments. Segments will go through TCP output engine and end up to tcp_transmit_skb().

tcp_write_xmit() takes care that segment is sent only when it is allowed to. If congestion control, sender window or Nagle's algorithm [7] prevent sending, the data will not go forward. Also retransmission timers will be set from tcp_write_xmit(), and after data send, congestion window will be validated referring to RFC 2861 [5].

tcp_transmit_skb() builds up TCP headers and passes data to network layer by calling function queue_xmit() found from **struct inet_connection_sock** from member **icsk_af_ops**.

3. ALGORITHMS AND OPTIMIZATIONS

This section will go through a few crucial parts of implementation and clarify why these are important features to have and to work properly in a modern TCP implementation.

3.1 TCP state machine

There are several state machines implemented in Linux TCP. Probably most known TCP state machine is connection state machine, introduced in RFC 793 [8]. Figure 3.1 presents states and transitions implemented in kernel. In addition to connection state machine TCP has own state machine for congestion control.

Majority of TCP states are handled in tcp_rcv_state_process(), as it handles all the states except ESTABLISHED and TIME_WAIT. TIME_WAIT is handled in tcp_v4_rcv(), and state ESTABLISHED in tcp_rcv_established(). From the viewpoint of user, ESTABLISHED is the most important state as the actual data transmission happens in that state. Therefore, tcp_rcv_established() is the most interesting and also likely the most optimized function in the TCP implementation. Implementation of it is divided into two parts: slow and fast path (section 3.3).

As stated, TIME_WAIT is handled in tcp_v4_rcv(). Depending on return value of tcp_timewait_state__process, packet will be discarded, acked or processed again with a new socket (if the packet was SYN initializing a new connection). Implementation of function is very clean and easy to follow.

3.2 Congestion control

At first TCP did not have specific congestion control algorithms, and due to misbehaving TCP implementations Internet had first 'congestion collapse' in October 1988. Investigation on that leaded to first TCP congestion control algorithms described by Jacobson in 1988 [6]. However, it took almost 10 years before official RFC based on Jacobson's research on congestion control algorithms came out [9].

Main file for TCP congestion control in Linux is tcp_cong.c. It contains congestion control algorithm database, functions to register and to active algorithm and implementation of TCP Reno. Congestion algorithm is linked to rest of the TCP stack by using **struct tcp_congestion_ops**, that has function pointers to currently used congestion control algorithm implementation. Pointer to the structure is found in **struct inet_connection_sock** (member **icsk_ca_ops**), see it at Figure 1.

Important fields for congestion control are located in **struct tcp_sock** (see section 2.1.1). Being the most important

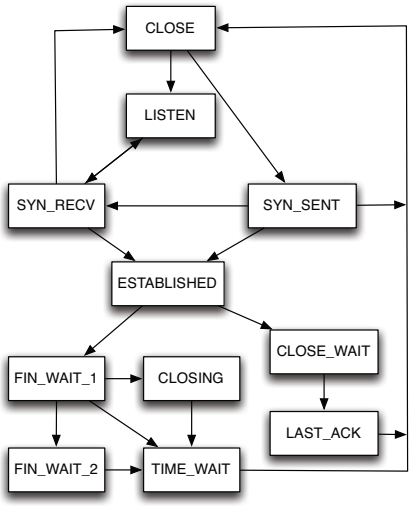

Figure 7: TCP connetion state machine in Linux kernel

variable, member **snd_cwnd** presents sending congestion window and **rcv_wnd** current receiver window. Congestion window is the estimated amount of data that can be in the network without data being lost. If too many bytes is sent to the network, TCP is not allowed to send more data before an acknowledgment from the other end is received.

As congestion control is out of scope of this paper, it will not be investigated more deeply.

3.3 TCP fast path

Normal, so-called slow path is a comprehensive processing route for segments. It handles special header flags and out-of-order segments, but because of that, it is also requiring heavy processing that is not needed in normal cases during data transmission.

Fast path is an TCP optimization used in tcp_rcv_established() to skip unnecessary packet handling in common cases when deep packet inspection is not needed. By default fast path is disabled, and before fast path can be enabled, four things must be verified: The out-of-order queue must be empty, receive window cannot be zero, memory must be available and urgent pointer has not been received. This four cases are checked in function tcp_fast_path_check(), and if all cases pass, will fast path be enabled in certain cases. Even after fast path is enabled, segment must be verified to be accepted to fast path.

TCP uses technique known as header prediction to verify segment to fast path. Header prediction allows TCP input machine to compare certain bits in the incoming segment's header to check if the segment is valid for fast path. Header prediction ensures that there are no special conditions requiring additional processing. Because of this fast path is easily turned off by setting header prediction bits to zero, causing header prediction to fail always. In addition to pass header prediction, segment received must be next in order to be accepted to fast path.

4. CONCLUSION

Implementation of TCP in Linux is a complex and highly optimized to gain as high performance as possible. Because of that it is also time-consuming process to get into code level in kernel and understand TCP details. This paper described the most fundamental components of the TCP implementation in Linux 3.5.3 kernel.

5. REFERENCES

[1] Linux kernel options documentation. `http://lxr.linux.no/#linux+v3.5.3/drivers/dma/Kconfig`.

[2] V. Cerf, Y. Dalal, and C. Sunshine. Specification of Internet Transmission Control Program. RFC 675, Dec. 1974.

[3] V. G. Cerf and R. E. Khan. A protocol for packet network intercommunication. *IEEE TRANSACTIONS ON COMMUNICATIONS*, 22:637–648, 1974.

[4] K. Cho, K. Fukuda, H. Esaki, and A. Kato. Observing slow crustal movement in residential user traffic. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 12:1–12:12, New York, NY, USA, 2008. ACM.

[5] M. Handley, J. Padhye, and S. Floyd. TCP Congestion Window Validation. RFC 2861 (Experimental), June 2000.

[6] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, Aug. 1988.

[7] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, Jan. 1984.

[8] J. Postel. RFC 793: Transmission control protocol, Sept. 1981.

[9] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001 (Proposed Standard), Jan. 1997. Obsoleted by RFC 2581.

TCP's Congestion Control Implementation in Linux Kernel

Somaya Arianfar
Aalto University
Somaya.Arianfar@aalto.fi

ABSTRACT

The Linux kernel implements various parts of the network stack. This paper is part of a joint attempt to describe the structure and the implementation details of the kernel code. In this specific paper, we explain parts of the Linux kernel code that deals with TCP's congestion control implementation. For this description we use the Linux kernel v 3.6.6. Our main focus is describing the most common pieces of the congestion control related code, which includes the congestion control framework itself, the interface between congestion control framework and rest of TCP, recovery state machine, and details of an example congestion control algorithms, TCP Cubic.

1. INTRODUCTION

The vast majority of the bytes on the Internet today are transmitted using Transmission Control Protocol (TCP) [7]. As a transport protocol, TCP is expected to provide support for different functionalities such as segmentation, reliability, and congestion control.

TCP's Congestion Control[6] is used to prevent congestion collapse[6, 9] in the network. To achieve this goal, TCP uses two basic elements: Acknowledgments (Ack) and Congestion Window. Acknowledgments are used to acknowledge reception of data by the receiver. Congestion Window is used to estimate the bottleneck capacity and control the maximum data that can be unacknowledged and on the fly in a connection.

The congestion control logic in TCP, basically uses the Additive Increase Multiplicative Decrease (AIMD) [1] model for capacity probing. In basic AIMD every acknowledgment results to an increase of maximum MSS (Maximum Segment Size) bytes to the congestion control window, while once per Round Trip Time (RTT) every loss results to reducing the congestion control window to half.

There are two basic phases in the AIMD algorithm: slow start and congestion avoidance. Slow start is usually used at the beginning of a connection. At the slow start phase the congestion control window increases exponentially. After the congestion window size reaches a predefined threshold (`ssthresh`), the algorithm enters the congestion avoidance phase. During the congestion avoidance congestion window's size doubles once per RTT, at maximum.

Some other concepts have been added to the basic congestion control algorithm [6] later on and then became part of the original algorithm in the form of New Reno[3] algorithm. Some of these added features include: Selective ACKs (SACKs) [1], Forward Acknowledgments (FACKs)[8], fast retransmit, and fast recovery. For more information regarding these features, the interested reader is referred to the IETF standard document on TCP's congestion control[1]. Additionally, throughout the years various other competing congestion control algorithms have been developed for TCP. Some of these algorithms include: Vegas[2], BIC[11], and Cubic[5].

These different congestion control algorithms for TCP, each apply their own tweaks to the basic AIMD model for better performance. Many of these algorithms are also implemented in Linux kernel. The original congestion control algorithm (New Reno) remains wired to the kernel code, while other algorithms could be plugged in, as we will describe later.

In this paper we attempt to describe part of the Linux kernel code (v 3.6.6) that deals with the TCP congestion control. We start by describing the code structure and relevant files in the implementation. We then continue by describing the information flows between different function and explain the approach taken in the kernel to implement the algorithmic details.

2. THE CODE STRUCTURE

Linux kernel implements TCP and its different congestion control algorithms. Before going into the kernel implementation details, it is important to note that congestion control and reliability are intertwined functionalities both in TCP's abstraction and in its kernel implementation. Therefore, the congestion control related code in kernel v.3.6.6 could be conceptually divided into 4 different categories: the congestion control framework itself, interface between congestion control framework and rest of TCP, recovery state machine, and details of different congestion control algorithms. Here we are going to briefly describe main data structures and related files used in TCP congestion control implementation.

2.1 Important Data Types

2.1.1 tcp_ca_state

TCP's congestion control implementation uses a state machine to keep and switch between different states of a connection for recovery purposes. These different states are defined in an enum type in tcp.h.

`Open`: When a connection is `Open` it is in a normal state,

with no dubious events, therefore packets received at this state go through the fast path. TCP fast path eliminates the extra processing that is required on flagged packets or in the case of suspicious loss or out-of-order delivery.

`Disorder`: This state is very similar to `Open` but requires more attention. It is entered when there are some SACKs or dupACKs. In this state some of the processing moves from fast path to the slow path.

`CWR`: State `CWR` is entered to handle some Congestion Notification event, such as ECN or local device congestion.

`Recovery`: This state shows that the congestion window has been reduced, and the connection is fast-retransmit stage.

`Loss`: State `Loss` shows that congestion window was reduced due to RTO timeout or SACK reneging.

2.1.2 tcp_congestion_ops

TCP congestion handler interface for different pluggable congestion control algorithms is described in `struct tcp_congestion_ops`, which is a structure of function call pointers. This structure is defined in tcp.h file.

```
struct tcp_congestion_ops {
      struct list_head  list;
      unsigned long flags;
     /* initialize private data (optional) */
     void (*init)(struct sock *sk);
      /* cleanup private data (optional) */
      void (*release)(struct sock *sk);
      /* return slow start threshold (required) */
      u32 (*ssthresh)(struct sock *sk);
      /* lower bound for congestion window
          (optional) */
      u32 (*min_cwnd)(const struct sock *sk);
      /* do new cwnd calculation (required) */
      void (*cong_avoid)(struct sock *sk, u32
          ACK, u32 in_flight);
      /* call before changing ca_state (optional)
          */
      void (*set_state)(struct sock *sk, u8
          new_state);
      /* call when cwnd event occurs (optional) */
      void (*cwnd_event)(struct sock *sk, enum
          tcp_ca_event ev);
      /* new value of cwnd after loss (optional)
          */
      u32 (*undo_cwnd)(struct sock *sk);
      /* hook for packet ACK accounting
          (optional) */
      void (*pkts_acked)(struct sock *sk, u32
          num_acked, s32 rtt_us);
      /* get info for inet_diag (optional) */
      void (*get_info)(struct sock *sk, u32 ext,
          struct sk_buff *skb);
      char name[TCP_CA_NAME_MAX];
      struct module *owner;
};
```

Some of the most important function calls in this structure are as follows:

`init()`: This function is called after the first acknowledgment is received and before the congestion control algorithm is called for the first time.

`pkts_acked()`: An acknowledgment that acknowledges some new packets, results to a call to this function. Number of packets that are acknowledged by this acknowledgments is paseed through the `num_acked` argument.

`cong_avoid()`: This function is called every time an acknowledgment is received and the congestion control state allows for congestion window to increase.

`undo_cwnd()`: returns the congestion window of a flow, after a false loss detection (due to false timeout or packet reordering) is confirmed.

2.2 Files

The main files that deal with the TCP code in the kernel are listed here. Many of these files could be found under `net/ipv4/` directory in the Linux kernel code, unless it is mentioned otherwise.

tcp.h: this files includes the TCP related definitions, including the data structures defined above. This file exist both in `include/net/` and `include/linux/` directories.

tcp.c: includes general TCP code and covers the interface between different sockets and the rest of the TCP code .

tcp_input.c: this is the biggest and most important file dealing with incoming packets from the network. It also contains the code for recovery state machine.

tcp_output.c: this files deals with sending packets to the network. It contains some of the functions that are called from the congestion control framework.

tcp_ipv4.c: IPv4 TCP specific code. This function hands the relevant packets to the congestion control framework.

tcp_timer.c: implements timer management functions.

tcp_cong.c: implements pluggable TCP congestion control support and congestion control's core framework with default implementation of New Reno logic.

tcp_[name of algorithm].c: these files implement different algorithm specific congestion control logic. For example, tcp_vegas.c implements the Vegas logic and tcp_cubic.c implements the TCP Cubic.

3. INFORMATION FLOW FOR THE RECOVERY STATE MACHINE

In this section, we describe what happens after a TCP connection is established and data and acknowledgment packets are exchanged. Adjustment of the congestion window and transition through the recovery state machine mainly depends on the reception of ACKs, or specific signs of congestion like timeouts and Explicit Congestion Notification (ECN [10]) bits. Simple form of TCP signals congestion by packet drops. ECN, however, allows for congestion notification without dropping packets. In case of congestion, an ECN-aware router can mark in the IP header instead of dropping the packet. The receiver of the packet echoes back the congestion indication to the sender by setting ECN_Echo (ECE [10])) flag in the TCP header.

Our main focus here is on handling received packets. But first we briefly describe countermeasures that sender of a

data packet takes to be able to handle ACKs and recognize congestion later on.

3.1 Recovery Handling Countermeasures

The main elements of handling ACKs and recognizing congestion on the data sender side are `retransmission queue` and `retransmission timer`. Transmission of a data packet is always followed by placing a copy of that data packet in a retransmission queue. The reception of an ACK then results to removing related copies in the retransmission queue. In current kernel the retransmission queue is defined as a member of `struct sock` and under the name `write queue`.

Each time a data packet is sent a retransmission timer is set for that packet. This timer counts down over time. In basic scenario, a packet is considered to be lost if its retransmission timer expires before an acknowledgment is received for that packet. In that case lost packets are retransmitted.

Anyhow, there are three tag bits, to mark packets in retransmission queue: SACKED (S), RETRANS (R) and LOST (L). Packets in queue with these bits set are counted in variables `sacked_out`, `retrans_out` and `lost_out`, correspondingly. While calculating the proper value for `retrans_out` for counting the number of retransmitted packets is pretty straight forward, marking right set of packets and calculating proper values for `sacked_out` and `lost_out` are a bit more complicated.

`sacke_out` counts the number of packets, which arrived to receiver out of order and hence not ACKed. With SACKs this number is simply amount of SACKed data. Without SACKs this is calculated through counting duplicate ACKs.

For marking the lost packet and calculating the `lost_out`, there are essentially two algorithms:

- FACK: Before describing the FACK algorithm to calculate the `lost_out`, we introduce another variable called `fackets_out`. In the code, `fackets_out` includes both SACKed packets up to the highest received SACK block so far and holes in between them. As soon as the FACK algorithm decides that something is lost, it decides that *all* not SACKed packets until the most forward SACK are lost. I.e.`lost_out = fackets_out - sacked_out` and `left_out = fackets_out`. It seems to be a correct estimate, if network does not reorder packets. But reordering can invalidate this estimation. There, the implementation uses FACK by default until reordering is suspected on the path. Reordering is often suspected with the arrival of duplicate Acks and SACKS.

- NewReno: when Recovery is entered, the assumption is that one segment is lost (classic Reno). When the connection is in Recovery state and a partial ACK arrives, the assumption turns to be that one more packet is lost (NewReno). This heuristics are the same in NewReno and SACK.

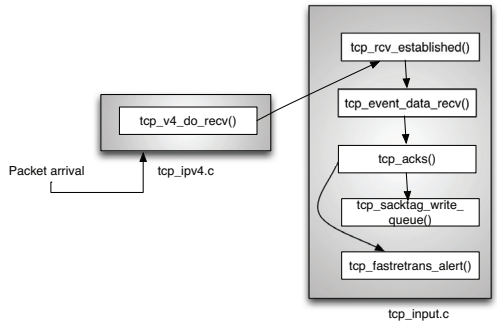

Figure 1: Packet reception to recovery state machine

Within TCP's retransmission logic[1]: with occurrence of the retransmission timeout, the TCP sender enters the retransmission timeout recovery where the congestion window is initialized to one segment and the whole window, which remains unacknowledged, is retransmitted. Therefore, After a RTO (retransmission timeout), when the whole queue is considered as lost, `lost_out` equals `packets_out`.

3.2 Recovery State Machine

As noted earlier, functions in tcp_input.c deal with the received packets. Therefore, the function calls we describe here are mostly from tcp_input.c unless mentioned otherwise.

As can be seen in Figure 1, reception of a data packet at the end-host triggers a call to `tcp_event_data_recv()`. This function in itself is called by `tcp_rcv_established()` which in turn called by `tcp_v4_do_rcv()` in tcp_ipv4.c.

The call to `tcp_event_data_recv()` results to measuring the MSS and RTT, and triggers an ACK (or SACK). The acking process benefits from two modes of operation:

- Quick ACK: It is used at the beginning of a TCP connection so that the congestion window can grow quickly.

- Delayed ACK: A connection can switch to this mode after a while. In this case an ACK is sent for multiple packets.

TCP switches between these two modes depending on the congestion experienced. Per default the quick ACK mechanism is enabled and ACK packets are triggered instantly to raise the congestion window fast especially for bulk data transfers. After a while when congestion window has grown enough delayed ACKs could be used to reduce the excessive protocol processing.

Incoming ACKs are processed in `tcp_acks()`. In this function the sequence numbers are processed to clarify what are the required actions after receiving an ACK. For instance some of the proper reactions could be: cleaning the retransmission queue, marking the SACKed packets, reacting to duplicate ACKs, reordering detection, and advancing congestion window. Here, we describe two of the most im-

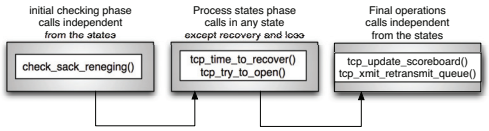

Figure 2: Recovery related functional calls in tcp_fast_retrans_alert()

portant functions that are called while processing ACKs in `tcp_acks()`.

3.2.1 tcp_sacktag_write_queue()

Incoming SACKs are processed in `tcp_sacktag_write_queue()`. In here `tcp_is_sackblock_valid()` tags the retransmission queue when SACKs arrive. This function is also used for sack block validation. SACK block range validation checks that the received SACK block fits to the expected sequence limits, i.e., it is between SND.UNA and SND.NXT. There is another function to limit sacked_out so that sum with lost_out isn't ever larger than packets_out.

3.2.2 tcp_fastretrans_alert()

The basic recovery logic and its related state transitions are implemented in `tcp_fastretrans_alert()` function. This function describes the Linux NewReno/SACK/FACK/ECN state machine and it is called from `tcp_acks()` in case of dubious ACKs. Dubious ACKs occur either when the congestion is seen for the first time or in other word the arrived ACK is unusual e.g. SACK, or when the TCP connection has already experienced something unusual that has caused it to move from the connection open state to any other state in the recovery state machine as described below and in Sec. **??**.

As can be seen in Figure. 2 different set of functions are called to check the state of a connection and do the proper operations at each state. The most important function calls executed in `tcp_fastretrans_alert()` are described in the following:

`tcp_check_sack_reneging()`: Packets in the retransmission queue are marked when a SACK is received (through another function as mentioned earlier). However, if the received ACK/SACK points to a remembered SACK, it probably relates to erroneous knowledge of SACK. `tcp_check_sack_reneging()` function deals with such erroneous situations.

`tcp_time_to_recover()`: This function checks parameters such as number of lost packets in a connection to decide whether its the right time to move to Recovery state. In other word, this function determines the moment when we decide that hole is caused by loss, rather than by a reorder. If it decides that is the recovery time; the CA State would switch to Recovery.

`tcp_try_to_open()`: If its not yet the time to move to recovery state, this function will check for switching the state and other proper reactions based on the indication in the packet. For example, if the ECE flag in the packet header is set, then the state will switch to CWR. Then, congestion window will be reduced by calling `tcp_cwnd_down`.

`tcp_update_scoreboard()`: This function will mark the lost packets. Depending on the choice of SACK or FACK all the packets which were not sacked (till the maximum seq number sacked) might be marked as lost packets. Also unacknowledged packets that have expired retransmission timers are marked as lost in this function. All the markings in this function triggers recounting for lost, sacked and left out packets.

`tcp_xmit_retransmit_queue()`: This function triggers retransmission of lost packets. It decides, *what* we should retransmit to fill holes, caused by lost packets.

`tcp_try_undo_<something>()`: The most logically complicated part of algorithm is undo heuristics. False retransmits can occur due to both too early fast retransmit (reordering) and underestimated RTO. Analyzing timestamps and D-SACKs can identify the occurrence of such false retransmits. Detection of false retransmission and congestion window reduction could be undone and the recovery phase could be aborted. This logic is hidden inside several functions named `tcp_try_undo_<something>`.

The functions above are mainly used for recovery state machine, and getting around the retransmission queue when there is a need for retransmission. However, we discuss the implementation for calculating the actual amount of increase/decrease in the congestion window size in the next section.

4. CONGESTION CONTROL ALGORITHMS

The basic congestion control functionalities core functionality is defined in `tcp_cong.cc`. TCP's original Reno algorithm[6] is directly implemented in `tcp_reno_cong_avoid()`. While in case of other algorithms, functions such as `tcp_slow_start()` and `tcp_cong_avoid_ai()` move the congestion window forward depending on the calculations done by different algorithm. These functions are called from different places in the code, for example from tcp_input.c or from any of the tcp_[name of algorithm].c files.

Figure 3 shows a high level abstraction of different congestion control algorithms are set and initialized in the code. Looking at the implementation from the users prospective, the only configurable part in this structure is the choice of congestion control algorithm. To achieve this goal, the implementation uses pluggable pieces of code in different files.

To register the pluggable congestion control algorithms, their implementation in different files such as tcp_vegas.c and tcp_cubic.c include a static record of `struct tcp_congestion_ops` to store and initialize the related function calls and algorithm's name. All these implementations register themselves into the system by calling (hooking to) the `tcp_register_congestion_control` from tcp_cong.c. However the algorithm used for every connection

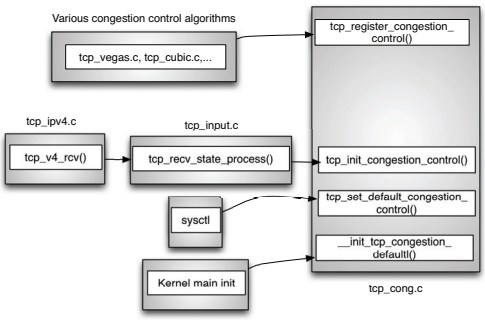

Figure 3: TCP congestion control related setting in Linux kernel (v 3.6.6)

is set up by kernel initialization or through a `sysctl` command. After the congestion control algorithm is set, defined hooks in `tcp_congestion_ops` are used to access the relevant algorithm specific functions from the rest of the code.

Implementation of all the algorithms more or less depends on the calculation of flight size and estimated size of the congestion window. Flight size shows the amount of data that has been sent but not yet cumulatively acknowledged. Therefore, it could be used to progress the congestion window, or estimate the correct value for e.g. sshtresh. In optimal situation, the flight size should be a reflection of bandwidth delay product.

In the code the flight size is shown through the `in_flight` variable.

$$in\_flight = packets\_out + retrans\_out - left\_out$$

In the equation above, `packets_out` is the highest data segment transmitted (SND.NXT) minus the first unacknowledged segment (SND.UNA) counted in packets. As shown in the equation, the estimation also needs to consider the number of retransmitted packets as part of the flight size calculation.

Theoretically, the sum of `retrans_out` and `packets_out` should show the flight size at any moment in time. However, in practice because of the usage of SACKs and other features, `packets_out` in its own reflects also those packets that have left the network in form of SACKed packets or lost packets, and thus are not in the flight anymore. In the equation above, `left_out` is number of these packets that left network, but not ACKed yet.

$$left\_out = sacked\_out + lost\_out$$

4.1 TCP Cubic in theory

The original Reno algorithm for TCP congestion control have been designed in those days when both the link capacities and round trip times were limited. Now a days, it is a known problem that as the bandwidth delay product grows TCP's sluggish behavior in increasing the congestion window size could result to under-utilization of network resource.

TCP cubic is one of the newest modifications to the TCP congestion control algorithm that changes the linear window growth function of TCP to a cubic function, in order to improve the bandwidth utilization in case of high bandwidth delay product networks. It also achieves a better level of fairness among flows with different round trip times. All these attributes make the cubic the default congestion control algorithm in Linux.

As it comes from the name of this algorithm the window growth function is a cubic function of elapsed time since the last packet loss. The algorithm registers a W_max to be the window size where the last packet loss event has happened. The algorithm then performs a multiplicative decrease of congestion window by a constant decrease factor. Afterwards, when the algorithm enters into congestion avoidance phase from fast recovery, it starts to increase the window using the cubic function that we will describe later. The cubic growth continues in a concave form until the window size becomes equal to the W_max. After that, the cubic function turns into a convex profile and the convex window growth continues from there. The concave-convex style of window growth helps the stability and better utilization of network resources [5].

4.2 TCP Cubic in the code

As mentioned earlier different pluggable congestion control algorithms are implemented in tcp_[name of algorithm].c files. They register themselves and their function calls to the system through initiating an instance of `tcp_congestion_ops`. One of these algorithms, which we are going to explain here, is TCP cubic. TCP cubic initiates its function calls in the code as follows:

```
static struct tcp_congestion_ops cubictcp
    __read_mostly = {
      .init       = bictcp_init,
      .ssthresh   = bictcp_recalc_ssthresh,
      .cong_avoid = bictcp_cong_avoid,
      .set_state  = bictcp_state,
      .undo_cwnd  = bictcp_undo_cwnd,
      .pkts_acked = bictcp_acked,
      .owner      = THIS_MODULE,
      .name       = "cubic",
};
```

After the initialization phase, `bictcp_acked()` is called on every received acknowledgment and triggers proper function calls for increasing/decreasing the congestion window. This function basically tracks delays and delayed acknowledgment ratio based on the following:

$$slidingwindowratio = (15 * ratio + sample)/16$$

The reason for tracking delayed ACKs is the logic im-

plemented in TCP cubic's code. In Cubic, congestion window is always increased if the ACK is okay, and the flow is limited by the congestion window. If the receiver is using delayed acknowledgement, the code needs to adapt to that problem.

TCP cubic's code integrates its own implementation for changing the congestion window size both at the slow start phase and at the congestion avoidance phase. Therefore, `bictcp_acked()` can also result to a call to `hystart_update()`. `hystart_update()` at the slow start phase increases the congestion control window based on the HyStart [4] algorithm instead of the standard TCP slow start logic. In this implementation HyStart logic is triggered when congestion window is larger than some threshold (`hystart_low_window __read_mostly = 16`).

Cubic Hystart uses RTT-based heuristics to exit slow start early on, before losses start to occur. Cubic HyStart use delays for congestion indication, but it exits slow starts at the detection of congestion and enters cubic's standard congestion avoidance.

For the congestion avoidance phase, the window growth function of TCP cubic [5] uses the following cubic equation:

$$CW(t) = C * (t - K)^3 + CW_max$$

where C is a CUBIC parameter, t is the elapsed time from the last window reduction, and K is the time period that the above function takes to increase current congestion window (CW) to CW_max when there is no further loss event and is calculated by using the following equation:

$$K = cubic_root(CW_max * beta/C)$$

where beta is the multiplication decrease factor (at the time of a loss window decreases to beta *CW_max).

In the code, `bictcp_cong_avoid()` is called during the congestion avoidance phase. The calculation of proper congestion window size at this stage is done based on the logic above and in `bictcp_update()` function. A summarized view of the function calls resulting to the ultimate final of the congestion window size, could be followed in the tcp_cubic.c file:

5. CONCLUSIONS

In this paper, we have described TCP's congestion control implementation in the Linux kernel v 3.6.6. Our main focus has been explaining the recovery state machine in the code and high level abstractions of congestion control's algorithm implementation.

6. REFERENCES

[1] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009.

[2] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. Tcp vegas: new techniques for congestion detection and avoidance. *SIGCOMM Comput. Commun. Rev.*, 24(4):24–35, 1994.

[3] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, 2004.

[4] S. Ha and I. Rhee. Taming the elephants: New tcp slow start. *Comput. Netw.*, 55(9):2092–2110, June 2011.

[5] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[6] V. Jacobson. Congestion avoidance and control. In *Proc. of ACM SIGCOMM '88*, volume 18, pages 314–329, Stanford, CA, USA, Aug. 1988. ACM.

[7] C. Labovitz, D. McPherson, S. Iekel-Johnson, J. Oberheide, F. Jahanian, and M. Karir. Internet Observatory Report. *Proc. NANOG-47*, 2009.

[8] M. Mathis and J. Mahdavi. Forward acknowledgement: refining tcp congestion control. *SIGCOMM Comput. Commun. Rev.*, 26(4):281–291, Aug. 1996.

[9] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, Jan. 1984.

[10] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, Sept. 2001.

[11] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *Proc. of IEEE INFOCOM 2004*, volume 4, pages 2514 – 2524, Hong Kong, March 2004.

Linux Implementation Study of Stream Control Transmission Protocol

Karthik Budigere
Department of Commnunication and Networking
Aalto University, Espoo, Finland
karthik.budigere@aalto.fi

ABSTRACT

The Stream Control Transmission Protocol (SCTP) is a one of the new general-purpose transport layer protocol for IP networks. SCTP was first standardized in the year 2000 as RFC 2960. SCTP is developed in complement with the TCP and UDP transport protocols. SCTP improves upon TCP and UDP and it also introduces new features such as multi-homing and multi-streaming, multi-homing feature provides the fault tolerance and the multi-streaming feature addresses the head-of-line blocking problem. This paper describes the high level details about the SCTP implementation in Linux kernel. It mainly focus on SCTP module initialization and registrations for socket layer abstraction, important data structures, SCTP sate machine and packet flow through SCTP stack.

1. INTRODUCTION

Stream control transmission protocol was designed to overcome the limitations of other transport layer protocols on IP such as TCP and UDP. The first SCTP specification was published in October 2000 by the Internet Engineering Task Force (IETF) Signaling Transport (SIGTRAN) working group in the now obsolete RFC 2960 [5]. Since then, the original protocol specification has been slightly modified (checksum change, RFC 3309 [6]) and updated with suggested implementer's fixes (RFC 4460 [4]). Both updates are included in the current protocol specification, RFC 4960 [3] that was released in September 2007. SCTP is rich in new features and capabilities. The capabilities of SCTP would make it suitable as a general transport protocol.

The following are some of the important new features of the SCTP; SCTP inherits some of the features from TCP along with some new exclusive new features.

- Multihoming: SCTP sends packets to one destination IP address and also has capabilities to reroute the messages using alternate route if the current IP address becomes unreachable. Hence SCTP offers resilience to the failed interfaces and faster recovery during network failures.
- Multi-streaming: SCTP supports multiple simultaneous streams of data with in the same connection. When sending the message systems cannot send parts of the same message through different streams; one message must go through one stream. In an ordered delivery option the stream is blocked when the packets are out of order or missing. Only one stream that is affected will be blocked where as the other streams will continue uninterrupted. Figure 1 shows the number of streams in an SCTP association.

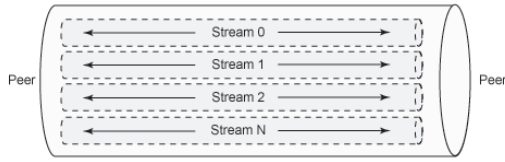

Figure 1: SCTP Multi Streaming Association

- Multiple Delivery Modes: SCTP supports multiple delivery modes such as strict order-of transmission (like TCP), partially ordered (per stream), and unordered delivery (like UDP). Message boundary preservation: SCTP preserves applications message boundaries by placing messages inside one or more SCTP data structures, called chunks. Multiple messages can be bundled into a single chunk, or a large message can be spread across multiple chunks.
- Selective acknowledgments. SCTP uses selective acknowledgment scheme, similar to TCP, for packet loss recovery. The SCTP receiver provides SACK to the sender with information regarding the messages to retransmit.
- Heartbeat keep-alive mechanism: SCTP sends heartbeat control packets to idle destination addresses that are part of the association. An SCTP association is nothing but the connection between the two endpoints. At a given time two endpoints will have only one SCTP association. The protocol declares the IP address to be down once it reaches the threshold of unreturned heartbeat acknowledgments.
- DOS protection. To avoid the impact of TCP SYN

flooding like attacks on a target host, SCTP employs a security cookie mechanism during association initialization. User data fragmentation: SCTP will fragment messages to conform to the maximum transmit unit (MTU) size along a particular routed path between communicating hosts.

The SCTP protocol can be divided into number of functions; these functions are Association startup and shutdown, Sequenced delivery of data within the streams, Data fragmentation, SACK generation and congestion control, chunk bundling, packet validation and path management. All these functions are discussed in detail in the following sections of the paper.

2. IMPLEMENTATION OVERVIEW

In this section we discuss the Linux implementation of SCTP. Basic operations of SCTP will be explained along with the description of the source code files, important fundamental data structures used by SCTP and we also describe the important functions in SCTP used in the packet reception from the network and to transmit packets back to the network.

The SCTP source code is present in the directory net/sctp and include/net/sctp. The important SCTP source files are listed in the table below.

File Name	Description
input.c	SCTP input handling, handles the incoming packets from IP layer
output.c	Hansles the outgoing SCTP packets and passes it on to the network layer
socket.c	SCTP extension for socket APIs.
sm_statetable.c	SCTP state machine table implementation
inqueue.c	SCTP input queue implementation
outqueue.c	SCTP output queue implementation
ulpqueue.c	SCTP uplink queue implementation, reponsible for transfering data to application
structs.h	All the important SCTP data structures such as association structure, end point structure are defined in this file

2.1 SCTP Initialization

SCTP is implemented as an experimental module in the Linux kernel. All the kernel modules have an initialization and exit functions. The SCTP kernel module initialization function is done in sctp_init(). This function is responsible for the initialization of the memory for several data structures and initialization various parameters used by the SCTP module. It also it performs the registration with socket and IP layer.

The important data structures that are initialized in this function are sctp_bucket_cachep of type sctp_bind_bucket, this data structure is used for managing the bind/connect, sctp_chunk_cachep that is of type struct sctp_chunk, this data structure is used to store the SCTP chunks. SCTP chunks are the unit of information within an SCTP packet, a chunk consist of a chunk header and chunk specific content, initializes SCTP MIB with SNMP (Simple Network Management Protocol), MIB (Management Information Base) are virtual database that are used for managing the entities in a communication network, setting up proc fs entry for SCTP by creating sctp directory under /proc (proc filesystem is used to access information about processes and other system information), initializes the stream counts and association ids handle, memory allocation and initialization of association and endpoint hash tables that are used in connection management, and initialization of SCTP port hash table.

2.1.1 Socket Layer Registration

The socket layer registration of SCTP is done by the functions sctp_v4_protosw_init() and sctp_v6_protosw_init(). The Linux kernel network subsystem data structures, struct proto defined in the file /include/net/sock.h and the struct net_proto_family defined in the /include/linux/net.h encapsulates the protocol family implementation. In order to register SCTP to TCP/IP stack (using IP as the network layer) we should Initialize an instance of struct proto and register to Linux network sub-system with call proto_register(). If the proto_register() function fails then the protocol addition with socket layer inetsw protocol switch will done by the function inet_register_protosw(), this function is defined in the net/ipv4/af_inet.c and takes argument of type proto structure defines the transport-specific methods (Upper layer) such as for connect (sctp_connect()), disconnect (sctp_disconnect()), sending (sctp_sndmsg()) and recieving (sctp_rcvmsg()) message etc, while the proto_ops structure defines the general socket methods.

2.1.2 IP Layer Registration

The IP layer registration or adding new transport protocol is performed in the function sctp_v4_add_protocol(). In this function it calls the function register_inetaddr_notifier in order to register to get notifications that notifies whenever there is addition or deletion of inet address and inet_add_protocol() function to add new protocol SCTP with inet layer. The function takes two arguments, first one is of type struct net_protocol in that we can specify the packet handler routine, for SCTP the packet handler routine is sctp_rcv() function and the second argument is protocol identifier that is IPPROTO_SCTP (132). This way the SCTP packets when received in the IP layer are sent to the sctp_rcv() function.

2.2 Data Structures

The following are some of the important data structures used in the SCTP Linux implementation,

- struct sctp_globals: The entire SCTP module universe is represented in an instance of struct sctp_globals. This structure holds system wide defaults for things like the maximum number of permitted retransmissions, valid

cookie life time, SACK timeout, send and receive buffer policy, several flag variables etc. It contains list of all endpoints on the system, associations on the system and also the port hash table.

- struct sctp_endpoint: Each SCTP socket has an endpoint, represented as a struct SCTP endpoint. The endpoint structure contains a local SCTP socket number and a list of local IP addresses. These two items defines the endpoint uniquely. In addition to endpoint wide default values and statistics, the endpoint maintains a list of associations. This logical sender/receiver of SCTP packets. On a multi-homed host, an SCTP endpoint is represented to its peers as a combination of a set of eligible destination transport addresses to which SCTP packets can be sent and a set of eligible source transport addresses from which SCTP packets can be received.

- struct sctp_association: Each association structure is defined by a local endpoint (a pointer to a sctp_endpoint), and a remote endpoint (an SCTP port number and a list of transport addresses). This is one of the most complicated structures in the implementation as it includes a great deal of information mandated by the RFC such as sctp_cookie, counts of various messages, current generated association share key etc. Among many other things, this structure holds the state of the state machine. The list of transport addresses for the remote endpoint is more elaborate than the simple list of IP addresses in the local endpoint data structure since SCTP needs to maintain congestion information about each of the remote transport addresses.

- struct sctp_transport: The struct sctp_transport defined by a remote SCTP port number and an IP address. The structure holds congestion and reachability information for the given address. This is also where we get the list of functions to call to manipulate the specific address family.

- struct sctp_packet: This is the structure which holds the information about the list of chunks along with the SCTP header information. These are getting assembled for the transmission. It has the destination information in the struct sctp_transport in it. An SCTP packet is a lazy packet transmitter associated with a specific transport. The upper layer pushes data into the packet, usually with sctp_packet_transmit(). The packet blindly bundles the chunks. It transmits the packet to make room for the new chunk. SCTP packet rejects packets that need fragmenting. It is possible to force a packet to transmit immediately with sctp_packet_transmit(). sctp_packet tracks the congestion counters, but handles none of the congestion logic.

- struct sctp_chunk This is the most fundamental data structure in SCTP implementation. This holds SCTP chunks both inbound and outbound. It is essentially an extension to struct sk_buff structure. It adds pointers to the various possible SCTP sub headers and a few flags needed specifically for SCTP. One strict convention is that chunk->skb->data is the demarcation line between headers in network byte order and headers in host byte order. All outbound chunks are ALWAYS in network byte order. The first function which needs a field from an inbound chunk converts that full header to host byte order. The structure also holds information about the sub headers present in the chunk and the sctp_transport information that tells source for an inbound chunk and destination for the outbound chunk.

2.3 Queues

There are four different queues in SCTP Linux implementation. They are sctp_inq, sctp_ulpq, sctp_outq, and sctp_packet. The first to carry information up the stack from the wire to the use and the second to carry information back down the stack. Each queue has one or more structures that define its internal data, and a set of functions that define its external interactions. All the queues have push inputs and external objects explicitly put things in by calling methods directly. A pull input is there for a queue and it would need to have a callback function so that it can fetch input in response to some other stimulus. These queue definitions are found in net/sctp/structs.h and net/sctp/ulpqueue.h.

2.3.1 sctp_inq

SCTP inqueue accepts packets from the IP layer and provides chunks for processing. It is responsible for reassembling fragments, unbundling, tracking received TSN's (Transport Sequence Numbers) for acknowledgement, and managing rwnd for congestion control. There is an SCTP inqueue for each endpoint (to handle chunks not related to a specific association) and one for each association.
The function sctp_rcv() (which is the receiving function for SCTP registered with IPv4) calls sctp_inq_push() to push packets into the input queue for the appropriate association or endpoint. The function sctp_inq_push() schedules either sctp_endpoint_bh_ rcv() or sctp_assoc_bh_rcv() on the immediate queue to complete delivery. These functions call sctp_inq_pop() to pull data out of the SCTP inqueue. This function does most of the work for this queue. The functions sctp_endpoint_bh_ rcv() and sctp_assoc_bh_rcv() run the state machine on incoming chunks. Among many other side effects, the state machine can generate events for an upper-layer-protocol (ULP), and/or chunks to go back out on the wire for transmission.

2.3.2 sctp_ulpq

sctp_ulpq is the queue which accepts events (either user data messages or notifications) from the state machine and delivers them to the upper layer protocol through the sockets layer. It is responsible for delivering streams of messages in order. There is one sctp_ulpq for every association. The

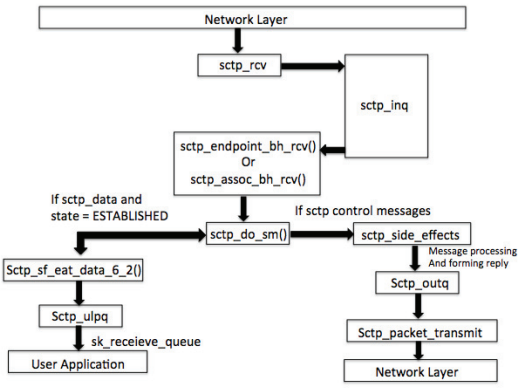

Figure 2: SCTP packet flow from network to userspace

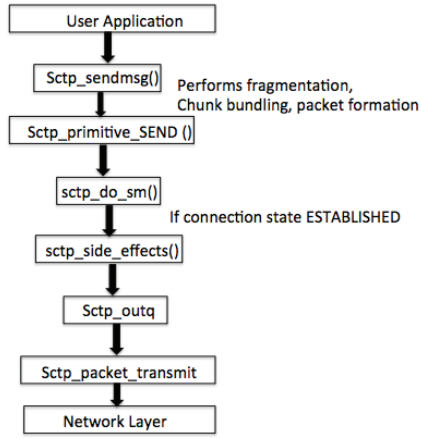

Figure 3: SCTP packet flow from user space

state machine, sctp_do_sm(), pushes data into an sctp_ulpq by calling sctp_ulpq_tail_data(). It pushes notifications with sctp_ulpq_tail_event(). The sockets layer extracts events from an sctp_ulpq with message written sk_data_ready() function sk_buff.

2.3.3 sctp_outq

sctp_outqueue is responsible for bundling logic, transport selection, outbound congestion control, fragmentation, and any necessary data queueing. It knows whether or not data can go out onto the wire yet. With one exception noted below, every outbound chunk goes through an sctp_outq attached to an association. The state machine injects chunks into an sctp_outqueue with sctp_outq_tail(). They automatically push out the other end through a small set of callbacks that are normally attached to an sctp_packet. The state machine is capable of putting a fully formed packet directly on the wire.

2.4 Packet Flow in SCTP

In this section we present the packet flow discussion of SCTP in the Linux kernel.

2.4.1 Packet flow from network

The packet flow from network to SCTP module is as shown in the Figure 2. The entry point for all the packets from the network layer to the SCTP module is the function sctp_rcv(). This is the function specified as the handler routine during the registeration with Network layer as discussed in the section 2.1.2.

Packets received from the network layer first undergoes the basic checks like checking for the minimum length of the packet received, checking for the out of the blue packets, etc. If the checks fail then the packet is discarded. All the proper packets received are pushed on to the sctp_inq for further processing. The packets are processed by the sctp_endpoint_bh_rcv() or sctp_assoc_bh_rcv() function that schedules packets from the sctp_inq. The processing of the packets are done by the state machine routine sctp_do_sm(). The state machine checks the type of chunks received and processes it accordingly. From the network layer we can either receive data or we can receive the SCTP control messages. If we receive data in the ESTABLISHED state then the data is passed on to the user application using the sctp_ulpq. If the packet received is one of the SCTP control message, then its processed according to the state and suitable action will be taken. If the state machine wants to reply the incoming message, then it forms the reply message and inserts the packet into sctp_outq for the transmission of the reply message to the network layer.

2.4.2 Packet flow from user space

The packets sent by the user space is sent to SCTP module using the function sctp_sendmsg() as shown in the figure 3. The packets are checked for the errors, if the packets are free from errors then we call sctp_primitive_SEND() for sending the packet. The SEND primitive calls the SCTP state machine for performing the suitable actions. The state machine function determines the suitable side effect action to perform and calls the sctp_side_effects() function for the command execution. If the packet is in order and error free then its inserted to the sctp_outq which will be scheduled by the function sctp_packet_transmit() function for transmitting the packet to the network layer.

2.5 Multihoming in SCTP

There are two ways to work with multihoming with SCTP. One way is to bind all your addresses through the use of INADDR ANY or IN6ADDR_ANY. This will associate the endpoint with the optimal subset of available local interfaces. The second way is through the use of sctp_bindx(), which

allows additional addresses to be added to a socket after the first one is bound with bind(), but before the socket is used to transfer or receive data. The multihoming is implemented in the function sctp_setsockopt_bindx() function. This function takes a argument op which specifies whether to add or remove the address from association. The binding completes by Sending an ASCONF (Address Configuration Change Chunk) chunk with Add IP address parameters to all the peers of the associations that are part of the endpoint indicating that a list of local addresses are added to the endpoint. If any of the addresses is already in the bind address list of the association, than we do not send the chunk for that association. But it will not affect other associations. The associations are created on the successful reception of the ASCONF_ACK chunk.

3. STATE MACHINE, ALGORITHMS AND OPTIONS

3.1 State machine

The state machine in SCTP implementation is quite literal. SCTP implementation has an explicit state table which keys to specific state functions that are tied directly back to parts of the RFC. The state machine table implementation can be found in the file sm_statetable.c. The handling functions are named uniquely as shown in the Table 1, function names contain the section of the RFC that it is refering, for example the function sf_do_5_1B_init is used for handling the INIT message in the CLOSED state and the numbering 5_1that is the suffix name in the function nae refers to the section 5.1 of the RFC 2960. All the state machine functions can be found in the file sm_statefuns.c. The core of the state machine is implemented in the function sctp_do_sm().
Each state function produces a description of the side effects (in the form of a struct sctp sm_retval) needed to handle the particular event. A separate side effect processor, sctp_side_effects() in the file sm_sideeffect.c, converts this structure into actions.
Events fall into four categories. The first category is about the state transitions associated with arriving chunks. The second category is the transitions due to primitive requests from upper layers, Not defined completely in the standards so its implementation specific. The third category of events is timeouts. The final category is a catch all for odd events like queues emptying. In order to create an explicit state machine, it was necessary to first create an explicit state table. Table 1 shows the partial state machine table with functions for different kind of chunks during different states.

3.1.1 SCTP connection Initiation

As SCTP and TCP are both connection oriented, they require communications state on each host. Two IP addresses and two port numbers define a TCP connection. An SCTP association is defined as [a set of IP addresses at A]+[Port-A]+[a set of IP addresses at Z]+[Port-Z]. Any of the IP addresses on either host can be used as a source or destination

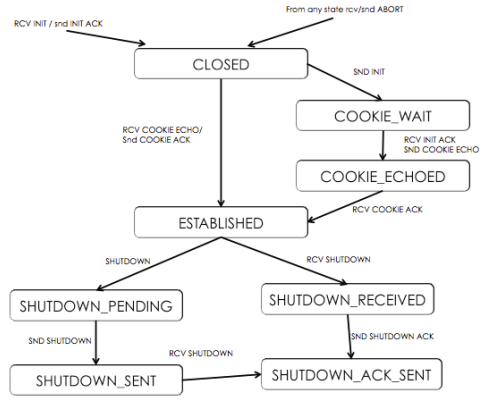

Figure 4: SCTP State Machine

in the IP packet and still properly identify the association. Before data can be exchanged, the two SCTP hosts must exchange the communications state (including the IP addresses involved) using a four-way handshake, a four-way handshake eliminates exposure to the aforementioned TCP SYN flooding attacks. The receiver of the initial (INIT) contact message in a four-way handshake does not need to save any state information or allocate any resources. Instead, it responds with an INIT-ACK message, which includes a state cookie that holds all the information needed by the sender of the INIT-ACK to construct its state. The state cookie is digitally signed via a mechanism. Both the INIT and INIT-ACK messages include several parameters used in setting up the initial state:

- A list of all IP addresses that will be a part of the association.
- An initial transport sequence number that will be used to reliably transfer data.
- An initiation tag that must be included on every inbound SCTP packet.
- The number of outbound streams that each side is requesting.
- The number of inbound streams that each side is capable of supporting.

After exchanging these messages, the sender of the INIT echoes back the state cookie in the form of a COOKIE-ECHO message that might have user DATA messages bundled onto it as well (subject to pathMTU constraints). Upon receiving the COOKIEECHO, the receiver fully reconstructs its state and sends back a COOKIE-ACK message to acknowledge that the setup is complete. This COOKIE-ACK can also bundle user DATA messages with it. The Table 1 gives information of which functions will be executed on receieving the messages based on the SCTP current state during the

Table 1: Partial SCTP state machine table

Message	CLOSED	COOKIE_WAIT	COOKIE_ECHOED	ESTABLISHED
SCTP_DATA	sf_ootb	Discard	Discard	sf_eat_data_6_2
INIT	sf_do_5_1B_init	sf_do_5_2_1_siminit	sf_do_5_2_1_siminit	sf_do_5_2_2_dupinit
INIT_ACK	sf_do_5_2_3_initack	sf_do_5_1C_ack	Discard	Discard
COOKIE_ECHO	sf_do_5_1D_ce	sf_do_5_2_4_dupcook	sf_do_5_2_4_dupcook	sf_do_5_2_4_dupcook
COOKIE_ECHO_ACK	Discard	Discard	sf_do_5_1E_ca	Discard
SCTP_HEARTBEAT	sf_ootb	Discard	sf_beat_8_3	sf_beat_8_3
SCTP_HEARTBEAT_ACK	sf_ootb	sf_violation	Discard	sf_backbeat_8_3

connection Initiation. The SCTP state is maintained by the sctp_association structure. Figure 4 shows how the SCTP state changes depending on the messages and received.

3.1.2 Fault Management

- End Point Failure Detection: SCTP keeps track of the total number of times the packet has been retransmitted to its peer consecutively. sctp_retransmit() function is used to retransmit the unacknowledged packets after the timer expiry. If the counter exceeds the limit then the peer endpoint is unreachable and the association enters the CLOSED state. The counter will be reset everytime when the DATA chunk is acknowledged by the peer endpoint.
- Path Failure Detection: HEARTBEAT message is used for the path management, everytime when a T3-rtx timer expires or when a HEARTBEAT message sent to an idle address is not acknowledged then the error counter for that destination address is incremented, if the counter exceeds the max value then the destination transport address is marked inactive. The counter is cleared when ever outstanding TSN is acknowledged or the when the HEARTBEAT ACK packet is received. sctp_do_8_2_transport_strike () function performs this path failure detection in the SCTP code. When the primary path is marked inactive the sender can automatically transmit new packets to an alternate destination address if it exists and is in the active state.
- Handling Out of blue packets: An SCTP packet is called an "out of the blue" (OOTB) packet if it is correctly formed, but the receiver is not able to identify the association to which this packet belongs. These packets are handled in the function sctp_rcv_ootb() as per the section 8.4 in the RFC 2960.
- Verification Tag: Every outbound SCTP packet contains the verification tag filled by the sending endpoint. The verification tag value is filled by the Initiate Tag parameter of the INIT or INIT ACK received from its peer. On receiving an SCTP packet, the endpoint should ensure that the value in the Verification Tag field of the received SCTP packet matches its own Tag. If the received Verification Tag value does not match the receiver's own tag value, the receiver shall silently discard the packet with some exceptions.

4. CONCLUSION

Despite a considerable amount of research, SCTP still lacks a killer application that could motivate its widespread adoption into the well-established IP networks protocol stack. Hence, SCTP is still not part of the vendor-supplied TCP/IP stack for widespread OSes[1]. One of the important milestones towards a broader adoption of SCTP was the decision within the mobile communications industry to select SCTP as a transport protocol for the Long Term Evolution (LTE) networks to support signaling message exchange between network nodes. SCTP is also the key transport component in current SIGTRAN suites used for transporting SS7 signaling information over packet-based networks. Hence, SCTP is used in progressively adopted Voice over IP (VoIP) architectures and thus becomes part of related signaling gateways, media gateway controllers, and IP-based service control points that are used to develop convergent voice and data solutions.[2]

5. REFERENCES

[1] Lukasz Budzisz, Johan Garcia, Anna Brunstrom, and Ferr. A taxonomy and survey of sctp research. *ACM Comput. Surv.*, 44(4):18:1–18:36, September 2012.

[2] Preethi Natarajan, Janardhan R. Iyengar, Paul D. Amer, and Randall Stewart. Sctp: an innovative transport layer protocol for the web. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 615–624, New York, NY, USA, 2006. ACM.

[3] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007.

[4] R. Stewart, I. Arias-Rodriguez, K. Poon, A. Caro, and M. Tuexen. Stream Control Transmission Protocol (SCTP) Specification Errata and Issues. RFC 4460 (Informational), April 2006.

[5] R. Stewart, Q. Xie, K. Morneault, and C. Sharp. RFC 2960, Stream control transmission protocol. http://www.faqs.org/rfcs/rfc2960.html, October 2000.

[6] J. Stone, R. Stewart, and D. Otis. Stream Control Transmission Protocol (SCTP) Checksum Change. RFC 3309 (Proposed Standard), September 2002. Obsoleted by RFC 4960.

The IPv4 Implementation of Linux Kernel Stack

Fida Ullah Khattak
Department of Communication and Networking
School of Electrical Engineering
Aalto University
fidaullah.khattak@aalto.fi

ABSTRACT

The foundations of modern day communication networks are based on the IPv4 protocol. Though specifications of IPv4 are readily available, this is not the case with its implementation details. Open source operating systems provide a chance to look at IPv4 implementation. However, because of their changing nature, the documentation of open source operating systems is often out-dated. This work is an attempt to document the implementation details of IPv4 in the Linux kernel 3.5.4.

Detail description of IPv4 packet traversal paths at the Linux kernel is provided. An overview of routing subsystem and its role in the IPv4 packet traversal is also given.

Keywords

Linux Kernel Stack, IPv4, Routing

1. INTRODUCTION

The linux IPv4 [3] implementation can be broken down into the input, the output and the forwarding paths. The IPv4 layer also interacts with supporting protocols, e.g. the ICMP [7], and subsystems, e.g. the routing subsystem, in the Linux stack for proper functioning.

An IPv4 packet that enters the IP layer, has to traverse through one of the paths mentioned earlier. Based on its origin and destination, the packet is routed to calculate the path of the datagram inside the Linux kernel. The functionalities of input, output and forwarding path traversal are implemented in their respective files as shown in table 1. It also shows the files that implements the routing functionality.

A packet inside the kernel is represented with an `sk_buff` structure. Apart from the datagram itself, this structure contains all necessary information required for routing the packet successfully. `sk_buff` is a common structure between all the layers of the IP stack. However, at a given time, only a single layer manipulates the values of this structure. In some cases, it is also possible to create multiple copies of the same structure for concurrent processing.

Table 2 shows some other structures important for packet processing at the IPv4 layer. Few of these structures are exclusive to the IPv4 layer, while others are shared by different layers of the TCP/IP stack.

Figure 1 gives an overall view of how the datagrams are processed at different paths. The traversal path has close interaction with the netfilter framework [2], which is used for functions like packet filtering and manipulation. Features like NAT [9] and firewalls [6] are implemented using the netfilter framework. As shown in Figure 1, there are several netfilter hooks included in the traversal logic for filtering packets at different points. Netfilter hooks pass control between different functions by taking a function pointer, called okfn, as an argument, which is called if a packet successfully traverses the hook.

The rest of the paper is structured as follows: Section 2 describes the ingress processing of IPv4 datagram. Section 4 covers forwarding while Section 5 explain the egress processing. Routing subsystem is explained in section6.

2. PROCESSING OF INGRESS IPV4 DATAGRAMS

The IPv4 ingress traversal logic, defined in the `ip_input.c` file, is responsible for delivering the datagrams to a higher layer protocol or forwarding it to another host. It consists of functions and netfilter hooks that process an incoming datagram. As shown by Figure 1, this path starts with the `ip_rcv()` function and ends at the `ip_local_deliver_finish()` function. The `ip_rcv()` function is registered as the handler function for incoming IPv4 datagrams at the system startup by the `ip_init()` routine.

Before the control is passed to `ip_rcv()`, the lower layer function, `netif_receive_skb()`, sets the pointer of the socket buffer to the next layer (layer 3), such that the start of `skb->hdr` is pointing to the IPv4 header. This way, the packet can be safely type-casted to an IPv4 datagram. We start with a brief description of each function at the input traversal path of IP layer, explaining how a datagram is processed on its way up the stack.

2.1 ip_rcv()

The first check performed by `ip_rcv()` is to drop the packets that are not addressed to the host but have been received by the networking device in promiscuous mode. It may also clone the socket buffer if it is shared with the net device. Next, `ip_rcv()` ensures that the IPv4 header is entirely present in the dynamically allocated area. Afterwards, the length, version and checksum values of the IPv4 header are verified.

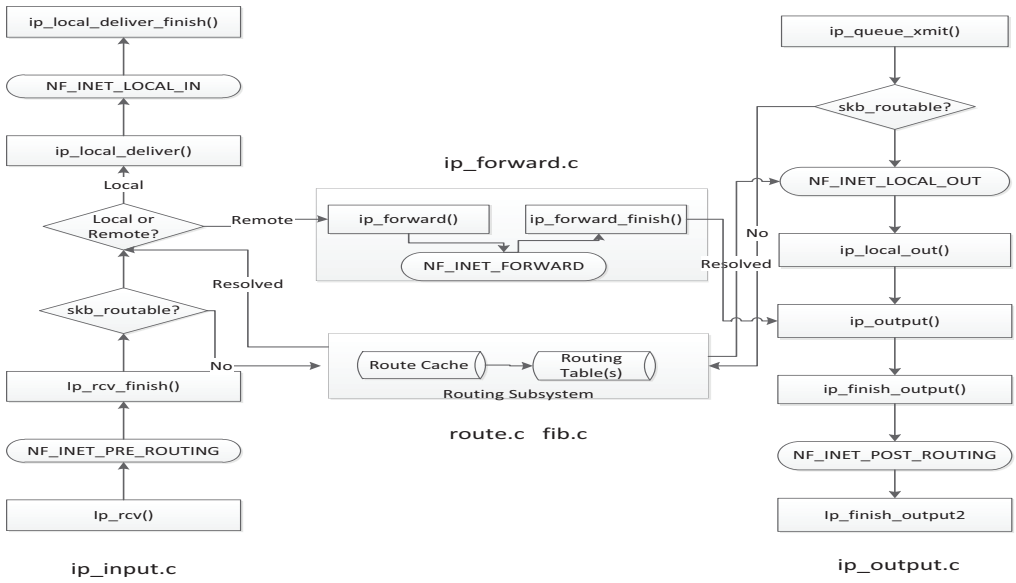

Figure 1: Traversal of an IPv4 datagram through Linux stack

Table 1: IPv4 Important files

File Name	Explanation
net/ipv4_input.c	Functions related to ingress path
net/ipv4_output.c	Functions related to egress path
net/ipv4_forward.c	Functions related to forwarding path
net/ipv4_fragment.c	Functions related to IP fragmentation
net/ipv4_route.c	Implementation of the IP_router

Once a packet has gone through all the sanity checks mentioned above, it is sent to the `NF_INET_PRE_ROUTING` hook. As explained in section 1, netfilter hooks provide a flexible framework to implement functions related to filtering and manipulation of IP datagrams. The `PRE_ROUTING` hook can be used to manipulate incoming datagrams before they have been routed. Thus all packets, local or remote, can be processed by handler functions associated with `PRE_ROUTING` hook. If a packet gets through this hook, the control is passed to `ip_recv_finish()` function.

2.2 ip_recv_finish()

This function is passed as a function pointer to `NF_INET_PRE_ROUTING` hook and is called when the packet has successfully passed the hook. It performs the key operation of routing the incoming datagrams using the `ip_route_input_no_ref()` function. More detailed description of the routing subsystem is given in the section 5 of this article.

If a packet is successfully routed, the routing subsytem initializes the route entry pointer in the socket buffer, `sk_buff->dst`, to a valid route from the routing subsystem. This valid route, apart from other information, contains a function pointer, skb->dst->input, initialized to one of the following functions:

1.`ip_forward()` function is selected if the destination is not the local host and the packet is to be forwarded to another host.

2.`ip_local_deliver()` function is selected if the packet is destined for the local host and has to be sent further up the stack.

3.`ip_mr_input()` function is selected if the packet is a multicast packet.

After successfully searching a valid route, IPv4 options are processed by calling `ip_rcv_options()` function. The last step of this routine is the call to the `dest_input()` function. This is a wrapper function which calls the `skb->dst->input()` function pointer, returned through the route lookup in the earlier step.

Table 2: IPv4 Related Data Structures

Data Structure	Location	Explanation
sk_buff	skbuff.h	Socket buffer data structure
flowi4	flow.h	Flow of traffic based on combination of fields
dst_entry	dst.h	protocol independent cached routes
iphdr	ip.h	IP header fields
ip_options	inet_sock.h	IP options of the header
ipq	ip_fragment.c	Incomplete datagram queue entry (fragment)
in_device	inet_device.h	IPv4 related configuration of network device

2.3 ip_local_deliver ()

This function is called when the routing entry returned in the route lookup operation points to the local delivery of datagram. The `ip_local_deliver()` routine is in charge of defragmentation. It calls the ip_defrag() function to queue the fragmented datagrams untill all the fragments have been received. After receiving a complete, unfragmented datagram,`ip_local_deliver()` calls the `NF_INET_LOCAL_IN` hook and passes `ip_local_deliver_finish()` as the function pointer to be called after the netfilter processing.

2.4 ip_local_deliver_finish ()

When a datagram successfully passes the netfilter hook called at the previous step, it is passed on to `ip_local_deliver_finish()` for some final processing at the IP layer. This function strips the IPv4 header from the `sk_buffer` structure and finds the handler for further processing based on the value of protocol field in the IPv4 header.

3. FORWARDING OF IPV4 DATAGRAMS

As shown in Figure 1, datagrams that reach the forwarding section have already been routed by the input path and do not require another routing lookup. This makes the forwarding path relatively simple in comparison to the input or the output path. Forwarding path is responsible for checking the optional features of "source routing" [1] and "router alert option" [5] before forwarding packets to other hosts. IPSEC policy checks for forwarded packets are also performed in the forwarding path.

Source routing option can be used by a sender to specify a list of routers through which a datagram should be delivered to the destination. Strict source routing implies that datagram must traverse all the nodes specified in the source routing list. If a datagram contains a "source routing option", the forwarding path checks if the next hop selected by the route lookup process matches the one indicated in the source routing list. In case the two hops are not the same, the packet is dropped and the source of the datagram is notified by sending an ICMP message .

Router alert option is used to indicate to the routers that a datagram needs special processing. This option is used by protocols like "Resource Reservation Protocol" [8] . A function can register itself as a handler for routing alert option, and if a datagram with a router alert option is received, the forwarding path calls this handler function. par We briefly take a look at the main functions, defined in `ip_forward.c` file, responsible for handling datagrams through the forwarding path.

3.1 ip_forward()

`ip_forward()` is the function that performs most of the forwarding related tasks, including the check for source routing and router alert options. Control is passed to the forwarding block when the routing logic decides that the packet is to be forwarded to another host and sets `skb->dst->input` pointer to the `ip_forward()` function.

Figure 2 shows the sequence of events in the `ip_forward()` function. It first checks for the IPSEC forwarding policies and the router alert option followed by checking the current TTL value of the datagram. If the router alert option is found, this function calls the corresponding handler responsible for implementing the route alert functionality and does not process the packet itself. If the TTL value of datagram is going to expire, an `ICMP TIME EXCEEDED` message is sent to the source and the packet is discarded.

Afterwards, the packet is checked for source routing. If the datagram contains a strict source routing option, it is made sure that the next hop, as mentioned in the source route list, is the same as calculated by the local route lookup. If not, the packet is discarded and source is notified.
At the end of the function, the netfilter hook, `NF_INET_FORWARD` is called , and ip_forward_finish() is passed as function pointer to the hook. This function is later executed if the packet successfully passes through the netfilter hook.

3.2 ip_forward_finish()

The `ip_forward_finish ()` function calls the `ip_forward_options()` function to finalize any processing required by the options included in the datagram and calculates the checksum.
At this stage the IPv4 header has been created and the datagram has successfully traversed all checks in the forwarding block. To transmit the packet, `ip_forward_finish()` calls `dst_output()` function, another function set by the routing subsystem during the ingress traversal.

4. PROCESSING OF EGRESS IPV4 DATAGRAMS

A higher layer protocol can pass data to IPv4 layer in different ways. Protocols like TCP [4] and SCTP [10], which fragment the packets themselves, interact with IPv4 layer for outgoing datagrams through `ip_queue_xmit()` function. Others protocols like UDP, that do not necessarily take care of the fragmentation, can call `ip_append_data()` and `ip_append_page()` functions to buffer data in Layer 3. This buffered data can then be pushed as a single datagram by calling the `ip_push_pending_frames()` function. TCP also

uses ip_build_and_send_pkt() and ip_send_reply() functions for transmitting SYN ACKs and RESET messages respectively.
However, as ip_queue_xmit() is the most widely used method for interacting with upper layer protocols, here we discuss the sequence of events when this function is called.

4.1 ip_queue_xmit()

ip_queue_xmit() is the main function of egress path which performs many critical operations on the datagram.Figure 3 shows some of the main ip_queue_xmit() operations. It is in the ip_queue_xmit() function that the outgoing datagram is routed and a destination is set for the packet.
The routing information required by a datagram is stored in skb->dst field. In some cases, it is possible that the outgoing datagram has already been routed (e.g. by SCTP) and the skb->dst contains a valid route. In that case, ip_queue_xmit() skips the routing procedure, creates the IPv4 header and sends the packet out.
In most cases, the skb->dst entry is empty and has to be filled by the ip_queue_xmit() function. There are three possible ways to route the packet.

1. The first option to find a valid route for an outgoing datagram is by using information from the "socket" structure. The socket structure is a part of sk_buffer structure and is passed as an argument to __sk_dst_check() function to search for an available route. If packets have already been sent by this socket, it will have a destination stored and can be used by any future packets originating from this socket.

2. Another way to find a route is through a cache lookup operation. If there is no route present in the socket structure sk_buffer->sk, the ip_queue_xmit() calls __ip_route_output_key() function for a lookup of a possible route in the routing cache.

This call is made through nested wrapper functions, ip_route_output_ports() and ip_route_output_flow(), which besides calling the __ip_route_output_key() function, perform IPSEC checks by calling the functions of XFRM framework and create a flowi4 structure. The flowi structure is later used by __ip_route_output_key() function for routing cache lookup.

3. If route cache lookup also fails, ip_route_output_slow() is called as a last resort to find a possible route by performing a lookup on the routing table known as the forwarding information base (FIB) .

After resolving the route, the ip_que_xmit() function creates the IPv4 header by using the skb_push() function. Ip_options_build() is called to build any IPv4 options. As the last step, dst_output() function is called. If no routes exist to the host, the packet is dropped while incrementing the IPSTATS_MIB_NOROUTES counter.

4.2 ip_local_out

This function is a wrapper for __ip_local_out() which computes the checksum for outgoing datagram after initializing the value for "IP header total length". It then calls the netfilter hook NF_INET_LOCAL_OUT, passing the dst_output() function as a function pointer. The NF_INET_LOCAL_OUT hook provides a control point to manipulate, filter etc. all outgoing traffic that is generated by the source itself.

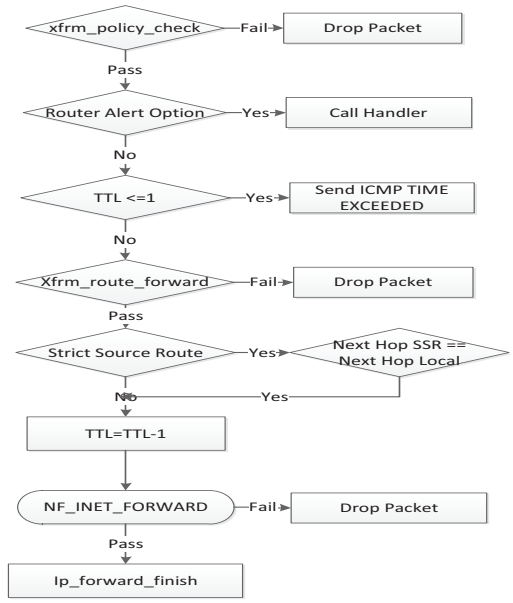

Figure 2: Forwarding of IPv4 Datagrams

4.3 dst_output()

Like the dst_input() function used for processing of incoming datagrams, dst_output() function is a wrapper for the function pointer skb->dst->output. This function pointer is set to a specific function when the route lookup operation is performed and skb->dst is initialized.In the case when the outgoing datagram is a unicast packet, the skb->dst->output is set to ip_output() function. For multicast packets, the value of skb->dst->output will point to the ip_mc_output() function. The routing operation is discussed in more detail in section 5.

If a packet successfully passes through the NF_INET_LOCAL_OUT hook called in the previous step, it is passed to the dst_output() function.At this point the datagram has been routed and its IP header is in place. The dst_output () function is called not only at the egress path but also at the forwarding path to transmit the outgoing packets. As the next step, this function invokes the function assigned to the skb->dst->output pointer.

4.4 ip_output()

This function is invoked for transmission of unicast datagrams. It is responsible for updating the stats of the outgoing packets for the network device and calls the netfilter hook NF_INET_POST_ROUTING . ip_finish_output() is passed as a function pointer to the hook.

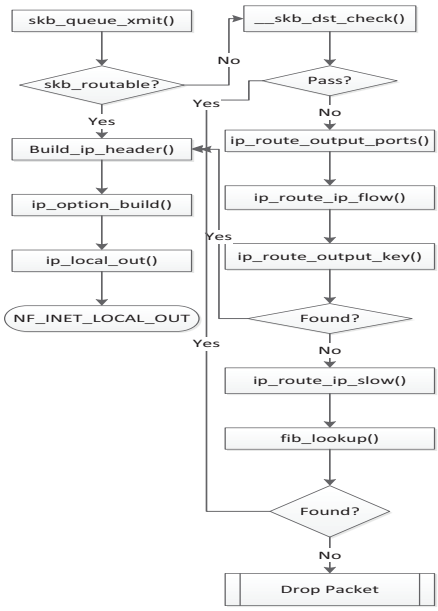

Figure 3: Egress Processing and Route Lookup

4.5 Ip_finish_output()

If the datagram traverses the `NF_INET_POST_ROUTING` hook successfully, it is passed to the `Ip_finish_output()` function for fragmentation checks. A packet with size more than the MTU is fragmented by calling the `ip_fragment()` function. When fragmentation is not required, the `ip_finish_output2()` function is called, which is the last hop for egress datagram processing at IP layer.

4.6 ip_finish_output2()

This function increments the counters for multicast and broadcast packets. It makes sure that the skb has enough space for the layer MAC header. It then tries to find the L2 neighbor address by searching for a prior cached entry for the destination. If that fails, it tries to resolve the L2 address by invoking the `neigh->output()` routine. After finding the L2 address, it calls the corresponding L2 handler for further processing. In case the search for L2 address fails, it drops the packet.

5. ROUTING SUBSYSTEM

The routing subsystem is essential to TCP/IP stack and it consists of a single routing cache and possibly many routing tables. The routing cache is a quick way for route resolution of datagrams. In case the route cache fails to provide an appropriate route entry, the routing tables are consulted to resolve the route.

Routing subsystem is initialized at the system startup by `ip_init()` function by calling `ip_rt_init()` routine. This routine initializes the routing cache by setting up the timers,

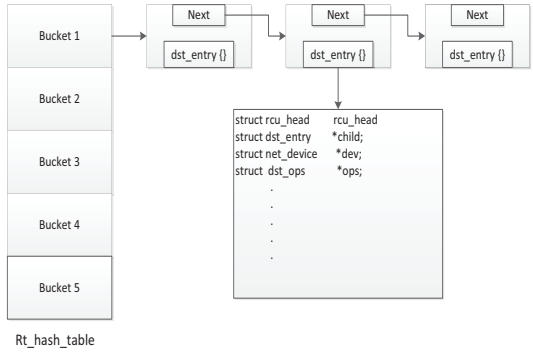

Figure 4: A simplified route cache

defining the size of the cache and starting the garbage collector. The routing table is initialized by calling the `_ip_fib_init()` and `devinet_init()` functions to register handlers and initialize the "fib" database.

Routing subsystem is used by both input and output traversal paths of the stack to "route" a datagram. A successful route search operation will return a valid route in the shape of `dst_entry` object, which is assigned to the `skb- >dst` pointer. This `dst_entry` object, apart from other information, contains two function pointers, `dst->input` and `dst->output`, which are initialized to point to a function based on the destination address of the route. The input and output traversal paths call these function pointers through their wrapper functions, `dst_input()` and `dst_output()`, respectively. As these function pointers can point to different handler functions, the flow of control can be directed based on the route entry. The assignment of one of the `ip_forward()`, `ip_local_deliver()` or `ip_mr_input()` to the `dst->input` function pointer, as shown in section 2.2, is an example of such control flow manipulation at the input traversal path.

5.1 Route Cache

A route cache lookup is the faster way to route packets in the Linux routing subsystem. The ip_route_input_no_ref() function of the incoming traversal path and the __ip_route_output_key() function of the output traversal path resolve routes for incoming and outgoing datagrams respectively, by performing the route lookup operation on the cache.

The route cache is implemented using the elements ofdst_entry structure, linked together to form "rt_hash_bucket". It is searched by using a "hash code" composed of the destination address, the source address, the TOS field values and the ingress or egress device. The computed "hash_code" is compared against the hash_code of the hash_buckets, and if a match is found, the entries in the bucket are compared against the values of flowi4 structure that is passed as an

argument to the route lookup operation.

Every time a route entry is successfully returned after a route lookup, a reference counter to the route cache entry is incremented . As long as the reference count is positive, the entry is not deleted. A garbage collection mechanism for routing cache makes sure that old and unused cache entries are deleted to create space for new entries in the cache. This mechanism can be invoked synchronously, if memory shortage is detected or asynchronously, through a periodic timer that checks for the expiry of route cache based on timers and reference counts.

5.2 Routing Table

The routing table is a complicated data structure. A default routing table has two tables, an"ip fib local table" and an"ip fib main table".
The former is used to keep routes to all local addresses whereas the latter is used for keeping routes to all other addresses. The entries in the routing tables are accessed through hash lookups, which provide an efficient search mechanism for matching route entries.

The IP layer functions use `ip_route_input_slow()` and `ip_route_output_slow()` functions to perform route lookups in the routing table, usually after route cache has failed. This function returns a pointer to routing table entry. Objects of `net` and `flowi4` structures are given as arguments. Information about the source address, destination address, possible output interface is gathered from the flowi4 structure. The type of service, input interface (as loopback by default) and and scope of the flow is gathered from the "net" argument and used for searching the route entries. The scope of the flow, RT_SCOPE_HOST, RT_SCOPE_LINK or RT_SCOPE_UNIVERSE indicates if the address belongs to the local machine, to a machine on the local network or to another machine that is not directly connected, respectively.

6. CONCLUSIONS

Even though IPv4 is an old and mature protocol, its kernel implementation is constantly changing due to additions and enhancements. In this paper, we discussed the implementation of IPv4 in a state of the art Linux kernel[1]. The input, output and forwarding paths were discussed in detail and the role of the routing subsystem in the routing of IP datagrams was explained.
It will be interesting to experiment with the size of routing cache and its impact on route lookup process as an extension to this work in the future.

7. REFERENCES

[1] Ip source route options. `http://www.juniper.net/techpubs/software/junos-es/junos-es92/junos-es-swconfig-security/ip-source-route-options.html`.

[2] The netfilter project. `http://www.netfilter.org/`.

[3] I. S. Institute. Internet Protocol. RFC 791, RFC Editor, September 1981.

[4] I. S. Institute. Transmission Control Protocol. RFC 793, RFC Editor, September 1981.

[5] D. Katz. IP Router Alert Option. RFC 2113, RFC Editor, February 1997.

[6] R. Oppliger. Internet security: firewalls and beyond. *Commun. ACM*, 40(5):92–102, May 1997.

[7] J. Postel. Internet Control Message Protocol. RFC 792, RFC Editor, September 1981.

[8] E. R. Braden, L. Zhang, and S. Berson. Resource Reservation Protocol. RFC 2205, RFC Editor, September 1997.

[9] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, RFC Editor, January 2001.

[10] Q. Xie, C. Sharp, and H. Schwarzbauerl. Stream Control Transmission Protocol. RFC 2960, RFC Editor, October 2000.

[1]version 3.5.4

Netfilter Connection Tracking and NAT Implementation

Magnus Boye
Aalto University School of Electrical Engineering
Department of Communications and Networking
P.O.Box 13000, 00076 Aalto, Finland
Email: firstname.lastname@aalto.fi

ABSTRACT

Good sources of information about the implementation of the Linux kernel are scarce. Due to the constant development, existing documentation quickly becomes outdated, although the general architecture of the kernel rarely changes radically. This paper attempts to give a detailed overview of the connection tracking and NAT modules in Netfilter. Understanding the architecture and implementation of these modules is necessary in order to modify or extend Netfilter. The architecture and implementation covered in this paper are based on kernel version 3.5.4.

Keywords

Linux kernel, Netfilter, connection tracking, NAT

1. INTRODUCTION

The small address space of IPv4 inevitably caused Network Address Translation (NAT) to be used in networks that are not assigned a public IP address range. In reality this is mostly residential Internet gateways where NAT offers multiple devices to share a single public IP address. It can be argued that NAT provides some level of security by hiding the structure of the LAN connected to the gateway. However, NAT is generally disliked in the networking community because it breaks the end-to-end principle, and IPv6 offers a solution to the problem that caused NAT to be used in the first place.

NAT is a stateful system that keeps track of incoming and outgoing flows of a network. NAT ensures that outgoing flows are mapped to a unique combination of IP address and transport-layer identifier on the external network. Figure 1 shows an illustration of source and destination NAT. Two client hosts A and B connect through a NAT gateway to an external network and have coincidentally selected the same source port for their outgoing flows. The NAT gateway performs source NAT on the outgoing flows by replacing the source IP address with the address of the gateway on the external network, and the source ports with available source ports associated with the IP address on the external network. Source NAT guarantees that a combination of internal network IP address and source port is mapped to a unique combination of external network IP address and source port. If such a mapping is not possible, traffic is dropped. When return traffic arrives at the NAT gateway, the destination IP address and destination port is replaced with the original source IP address and port. The gateway also performs destination NAT which is commonly called "port forwarding" in consumer routers. In figure 1, the gateway performs destination NAT on port 80 and 22 on incoming traffic from the external network. Destination NAT works by modifying the destination of incoming traffic, just like source NAT does for return traffic. In this scenario traffic to port 80 and 22 are directed to the hosts WWW and SSH on the internal network, respectively. Source NAT and destination NAT can be used simultaneously, as long as source NAT does not map any outgoing flows to ports used by destination NAT. Depending on the configuration, destination NAT can have various purposes, for instance load balancing.

In addition to transport-layer protocols, some protocols such as ICMP also use identifiers to distinguish between flows. The most common identifiers are 16 bit port numbers as used by TCP and UDP. The size of the protocol-specific identifier determines the number of connections the gateway can handle for the protocol. A 16 bit identifier theoretically allows for up to $2^{16}-1$ simultaneous flows through a gateway. The number of possible simultaneous flows can be increased by using NAT with multiple external IP addresses.

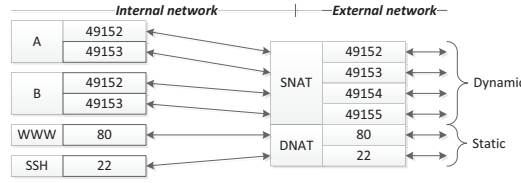

Figure 1: Source and destination NAT.

The book by K. Wehrle et al.[4] gives a broad and detailed overview of the Linux networking architecture and implementation. The book describes the architecture as of kernel version 2.4 which dates back to 2001. Many aspects of the architecture described in the book are similar with the kernel version 3.5.4, but the source code is far from the same. J. Engelhardt[1] is maintaining a short guide on how to create Netfilter modules. The guide gives a brief introduction to Netfilter and connection tracking, but does not describe the architecture and implementation of connection tracking and NAT in detail. The connection tracking and NAT modules support several transport-layer protocols. In order to give a more concise overview in this paper, it was decided to focus on the generic aspects of the modules, and UDP over IPv4.

This paper is structured as follows. Section 2 gives a short introduction to the Netfilter hooks used Netfilter modules. Section 3 gives an overview of the connection tracking module, its key data structure and functions. Section 4 gives an overview of the NAT module and how it relates to the connection tracking module. Section 5 describes potential vulnerabilities in the implementations of connection tracking and NAT.

2. NETFILTER FRAMEWORK

Netfilter is a framework for packet manipulation and filtering. The framework provides access to packets through five hooks in the Linux kernel at key points in packet processing. The hooks exist for both IPv4 and IPv6. Figure 2 shows in which order the Netfilter hooks are called when processing an IPv4 packet. The return value from a Netfilter hook must be one of five options: `NF_ACCEPT`, `NF_DROP`, `NF_STOLEN`, `NF_QUEUE`, `NF_REPEAT`. The return value is also referred to as a *verdict*. The first two options accept or drop a packet, respectively. If multiple functions are attached to a hook, the packet will be dropped if a single function returns `NF_DROP`. The return value `NF_STOLEN` indicates that a packet has been consumed by the hook function and further processing by other functions attached to the hook is not possible. `NF_QUEUE` indicates that the packet should be inserted into a queue, and `NF_REPEAT` indicates that the hook function should be called again. When functions are registered to hooks, a priority of the functions is given. This priority determines the order order in which the functions attached to the same hook are called. This paper focuses on events that mainly take place at the `NF_IP_PRE_ROUTING` and `NF_IP_POST_ROUTING` hooks, as these are key points in connection tracking and NAT.

Hooks are registered by calling `nf_register_hook()`, which takes a pointer to a `struct nf_hook_ops` as parameter. This structure defines the actual function called by the hook, address family, priority, and at which hook the function should be called as shows in Figure 2. The various hooks are integrated into the IPv4 implementation using two functions: `NF_HOOK()` and `NF_HOOK_COND()`. The former executes hooks and the later works just like `NF_HOOK`, except an execution condition can be given. Both functions are passed a pointer to `okfn()` which is the callback function for the hook point. The provided `okfn()` is called if the condition is `false` when using `NF_HOOK_COND()`, or if the final verdict of the functions registered to the hook is `NF_ACCEPT`. The functions in the IPv4 implementation where hooks are executed are shown in Table 1.

Figure 3 shows how functions registered to hooks are executed when `NF_HOOK()` is called. `nf_hook_thresh()` check if Netfilter hooks are enabled and then calls `nf_hook_slow()`, which calls the main function `nf_iterate()` and evaluates the return value of this function. As the name suggests, `nf_iterate()` iterates over the function registered to the hook specified when `NF_HOOK()` was called. `nf_iterate()` calls each registered function and checks their return values. If any of the functions return `NF_DROP`, the packet must be dropped and thus it is not necessary to call any remaining functions for the hook. If the function return `NF_REPEAT`, the function is called again in `nf_iterate()`, and if no functions have been registered for the hook `NF_ACCEPT` is returned. The return value of `nf_iterate()` is evaluated by `nf_hook_slow()`, mainly to perform potential queuing. Only the eight least significant bits of the return value are used to store one of the five possible Netfilter verdicts. The remaining bits in the 32 bit return value can be used for other data. If the Netfilter verdict is `NF_QUEUE`, the 16 most significant bits are used to indicate which Netfilter queue the packet should be inserted into. The queue number is passed to `nf_queue()` which queues the packet. When the packet has passed through the queue it will be re-injected into the packet processing flow by the function `nf_reinject()`. If the verdict returned by `nf_iterate()` is `NF_DROP`, the function `kfree_skb()` is called and the packet is dropped.

Hook	Caller
NF_INET_PRE_ROUTING	ip_rcv()
NF_INET_LOCAL_IN	ip_local_deliver()
NF_INET_FORWARD	ip_forward()
NF_INET_LOCAL_OUT	__ip_local_out()
NF_INET_POST_ROUTING	ip_output()

Table 1: Netfilter hooks in IPv4.

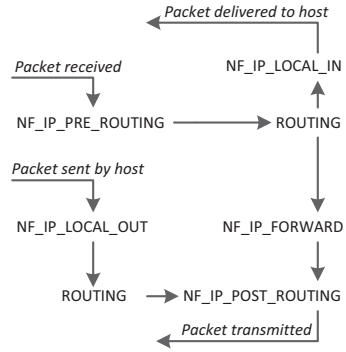

Figure 2: Netfilter IPv4 hook traversal.

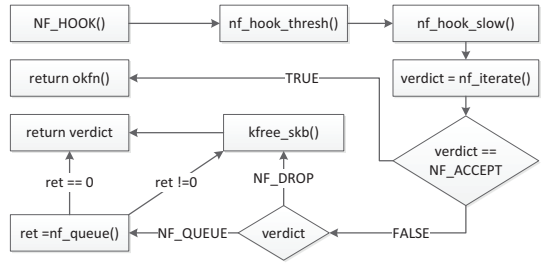

Figure 3: Hook execution by `nf_hook()`

3. CONNECTION TRACKING

The connection tracking(CT) module is responsible for identifying trackable packets belonging to trackable protocols. The module supports tracking of both stateless and statefull protocols. The CT module operates independently of the NAT module, but its primary purpose is to support the NAT module.

3.1 Tuples

The most important data structure of the CT module is `struct nf_conntrack_tuple`. This "tuple" structure is used to represent a unidirectional packet flow by its network-layer and transport-layer addresses. Bidirectional flows are thus represented using a tuple for each direction. Figure 4 shows a simplified representation of `struct nf_conntrack_tuple`. The data structure uses `unions` to contain both protocol-specific fields and generic fields in `dst.u`. This makes the source code easier to understand, optimizes memory, and allows new protocol-specific fields to be added without breaking the existing code. The `dst.u` field defines a `union` of 16 bit that contains fields for the following protocols: TCP, UDP, ICMP, DCCP, SCTP, and GRE. The fields reveal information about the header information in different protocols used to uniquely identify a packet flow. For instance, TCP and UDP use port numbers while ICMP uses ICMP type and code.

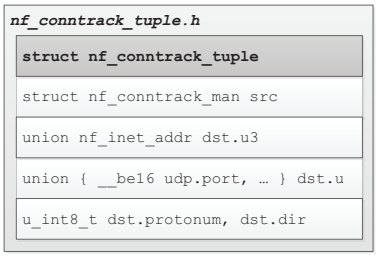

Figure 4: `struct nf_conntrack_tuple`

Since a tuple contains different information depending on both the network-layer protocol and transport-layer protocol of a packet, each supported protocol is implemented as a module. The modules conform to the interface defined by the two structures `struct nf_conntrack_l3proto` and `struct nf_conntrack_l4proto`. The structures contain function pointers that are initialized to the appropriate functions in the protocol-specific modules. Figure 5 shows the initialization values of the two structures for a UDP packet encapsulated by IPv4. The most important function pointer that both structures have in common is `pkt_to_tuple()`. This pointer points to a function which maps a packet to a tuple based on its network-layer or transport-layer data. In the case of IP, `pkt_to_tuple()` sets the `dst.u3` and `src.u3` fields of a tuple to the source and destination IP address of the packet, respectively. In the case of UDP, `pkt_to_tuple()` sets the `dst.u` and `src.u` fields to the source and destination UDP ports, respectively.

The `l3proto` and `l4proto` fields in Figure 5 are set to address family and protocol numbers as defined in the Linux kernel. Note that these values are not the same as specified by IANA[2][3], although some of them overlap. An explanation of the difference between Linux and IANA address families could not be found. Considering that IP was invented before IANA was founded, it is possible that the numbering of address families in Linux is simply a legacy from previous operating systems. The `get_timeouts()` function returns the timeout values of the protocol. The `error()` function checks for special packets that cannot be tracked, and `new()` is called when a new flow is seen by the CT module. Finally, `packet()` function is called for all packets which are deemed trackable by `error()`.

Constant	Description
`IPS_EXPECTED`	The connection was expected
`IPS_SEEN_REPLY`	Bidirectional traffic has been seen
`IPS_ASSURED`	Never expire connection prematurely
`IPS_CONFIRMED`	Packets were transmitted
`IPS_SEQ_ADJUST`	TCP needs sequence number adjustment
`IPS_DYING`	Connection is dying

Table 2: Connection status values.

3.2 Hashing

The CT module is optimized for performance and therefore stores the CT state of active connections in a hash table. The function `hash_conntrack_raw()` returns a generic 32 bit hash of a tuple. The hash value is based on the source and destination IP addresses and protocol-specific identifier. The `nf_conntrack_tuple_hash` structure is used to store a CT state in the hash table and contains the tuple along with a pointer to a linked list of CT state associated with the tuple. The linked list is used to handle hash collisions.

3.3 Connections

Netfilter uses the term *connection* even for packet flows in connectionless protocols. For the sake of clarity the term flow is used in this paper. A tracked flow is represented by a `struct nf_conn` which is shown in Figure 6. The `tuplehash` field contains a `struct nf_conntrack_tuple_hash` for each direction of the flow, and these structures contain a reverse pointer to the `nf_conn` structure. The key fields in the data structure are `timeout` and `status`. The `timeout` field contains a list of timers related to the connection state. These timers are necessary in order to optimize resource utilization. The CT states of inactive flows are removed to reduce memory and enable faster hash table lookups. Connection-oriented protocols have multiple states and the lifetime of CT states for such flows depend on the protocol state. The connection tracking module gives priority to established connections in the case of connection-oriented protocols and bidirectional flows in the case of connection-less protocols. For instance, the CT state of a TCP connection has a lower lifetime during the initial three-way handshake compared to after the handshake has been completed. In the case of UDP, unidirectional flows have a shorter lifetime than bidirectional flows, because the bidirectionality indicates that a flow is important. The `status` field is used as a bitset where different bits correspond to different protocol states, as specified by `enum ip_conntrack_status`. The most important connection states are shown in Table 2.

3.4 Tracking

The CT modules uses three Netfilter hooks to track incoming and outgoing packets. The function `nf_conntrack_in()` is called by the `NF_INET_PRE_ROUTING` and `NF_INET_LOCAL_OUT` hooks. The function `nf_conntrack_confirm` is called by the `NF_INET_POST_ROUTING` hook. The `nf_conntrack_in()` function is the main function of the CT module. The initial steps of the `nf_conntrack_in()` function is to determine the

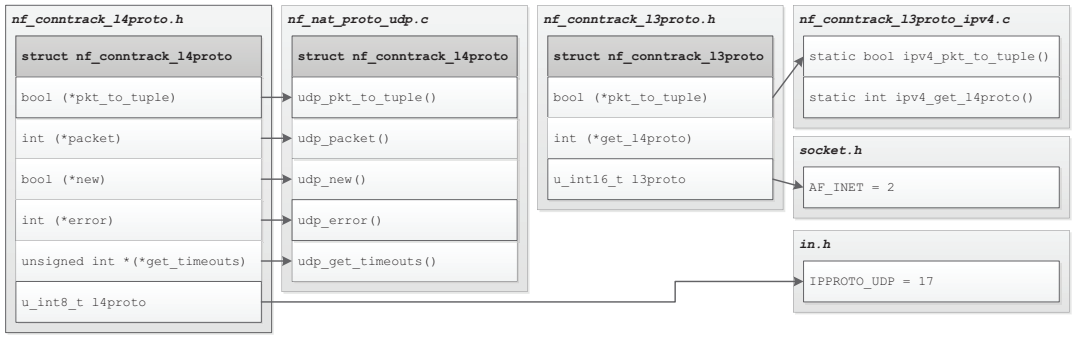

Figure 5: `struct nf_conntrack_l3proto` **and** `struct nf_conntrack_l4proto`

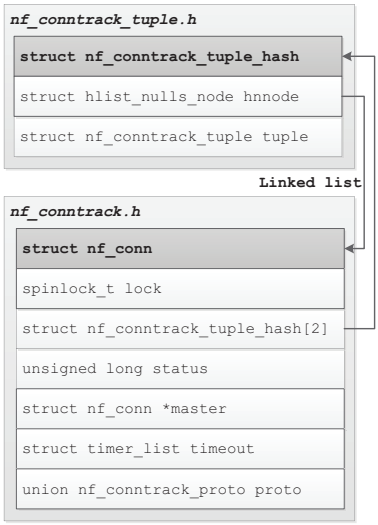

Figure 6: `struct nf_conntrack_tuple_hash` **and** `struct nf_conn`

network-layer protocol and transport-layer protocols. If the protocols can be tracked, a `struct nf_conntrack_l3proto` and a `struct nf_conntrack_l4proto` are initialized, as previously described. Before the main protocol-specific tracking functions are called, the `error()` function is called. In the case of UDP, the function checks for malformed packets with invalid payload size or invalid checksum. If the `error()` function returns `NF_ACCEPT` the packet is trackable and `resolve_normal_ct()` is called. This function ensures that a CT state exists for the packet tuple, by creating a new CT state if this is the first packet in a flow. The function begins by calling the protocol-specific `pkt_to_tuple()` function and obtains a tuple. The hash of the tuple or inverse tuple is calculated and used to retrieve a possibly existing CT state from the hash table of the CT module. If no CT state is found a new state is created by calling `init_conntrack()`. Otherwise the existing the CT state is returned.

The `init_conntrack()` function creates a new `nf_conn` structure and initializes its values by calling the protocol-specific function `new()`. The function continues by checking if the packet was expected as a result of protocol behavior in a flow tracked by another CT state. If the flow was expected, the `master` field of the `nf_conn` structure is set accordingly. Some application protocols utilizes multiple packet flows and in order for such applications to function properly, the NAT module must be able to direct incoming flows to the right host. Since protocol behavior varies from protocol to protocol, protocol-specific modules are needed to detect protocol behavior that results in new flows. For instance, in passive FTP the client tells the server to connect to the client on a specified address and port when a file transfer is requested. The CT module creates an *expected* CT state for the incoming flow, so that it can be identified and directed by the NAT module. The `struct nf_conntrack_tuple_hash` of the packet is inserted into a list of *unconfirmed* connections. If the packet is not dropped by any other Netfilter modules, the packet should be observed by `nf_conntrack_confirm` at the `NF_INET_POST_ROUTING` hook. The purpose of this function is to check that a packet belonging to a connection actually makes it onto the network and was not dropped by another module. If the packet is seen by this function, the state of the connection is changed to `IPS_CONFIRMED` and the connection is removed from the list of unconfirmed connections and inserted into the hash table of the CT module. After the `resolve_normal_ct()` function has ensured that a CT state exists, the function returns a pointer to the CT state of the packet.

`nf_conntrack_in()` continues by obtaining the protocol-specific timeout values and then calls the `packet()` which points to `udp_packet()` for the UDP protocol. Because UDP is connectionless, the connection tracking functions are not very advanced. The `udp_packet()` function simply extends the timeout of the connection based on whether the `IPS_SEEN_REPLY` bit has been set in the connection status. If bidirectional traffic has been seen, the connection timeout is extended further than if only unidirectional traffic has been seen. As mentioned earlier, the reason for this behavior is that the CT module optimizes resource consumption by inactive flows. The shorter timeout for unidirectional connections does not limit connectivity, but requires more frequent packet transmissions to prevent the flow from expiring. The

shorter timeout also makes the CT module more tolerant to denial-of-service attacks, although the default timeouts are still too high to mitigate attacks. The default timeout specified for UDP in the CT module is 30 seconds for a unidirectional (unreplied) connection, and 3 minutes for a bidirectional connection.

The `udp_packet()` function always returns `NF_ACCEPT` since screening for bad packets has already been performed by `udp_error()` and therefore nothing can go wrong in this function. The final verdict returned by `nf_conntrack_in()` is determined by `udp_error()` or `udp_packet()` function in the case of UDP.

4. NETWORK ADDRESS TRANSLATION

Although the Netfilter NAT module does modify the network-layer addresses of packets, it appears the addresses may be altered by other modules invoked after the NAT module. In particular the Masquerade module replaces the source IP address of packets such that they match that of the transmission interface. The module creates a NAT rule such that traffic in the reverse direction will have their destination address replaced with the original source address. This section covers transport-layer NAT. The inner workings of the Masquerade module and network-layer NAT is an area that needs further documentation.

The NAT module is comprised of a set of core functions that perform general NAT tasks, and several protocol-specific NAT modules. The main function of the NAT module is `nf_nat_fn()` which is called by four helper functions at the following Netfilter hooks: `NF_IP_PRE_ROUTING`, `NF_IP_POST_ROUTING`, `NF_IP_LOCAL_OUT`, and `NF_IP_LOCAL_IN`. The protocol-specific modules are needed because some protocols exchange address information at the application layer and this information needs to be altered according to the address modifications performed at the network and transport layers. The protocol-specific modules conform to the interface defined by `struct nf_nat_protocol`. The structure defines four function pointers and their relationship to the UDP NAT module is shown in Figure 7. The function `manip_pkt()` alters a packet based on a tuple and the type of NAT: source NAT or destination NAT. The function replaces the network-layer address and transport-layer address information in the packet with the information in the supplied tuple. The function `unique_tuple()` provides the tuple that is passed to `manip_pkt()`. The purpose of the function is to determine an available protocol-specific identifier on the external network. In the case of UDP the function `nf_nat_proto_unique_tuple()` is used to provide an available 16 bit port number. The function can be used by all protocols that use 16 bit port numbers and returns either a randomly selected and available port, or an available port from a specified range. The random port number is generated by inputting source address, destination address, destination port, and a random number into the MD5 algorithm. If a static port range has been specified, the port number is not selected randomly but from the beginning of the specified range. The `unique_tuple()` function updates the offset of the range each time it is called, and thereby ensures that returned port numbers increase monotonically within the range. The address range is provided by the `nlattr_to_range()` function. The `in_range()` function determines if a packet belongs to a group of packets that should be processed by the NAT module. If the address range is exhausted the NAT modules will begin to drop packets. The UDP NAT module utilizes the function `nf_nat_proto_in_range()`, which checks if the port number of a packet is within a range that should be processed by the NAT module. The function ensures that if a small range of port numbers is used by the NAT module, then the more complex NAT functions are only called if the packet might actually have been altered by the NAT module. If port numbers are chosen randomly the function will almost always return.

The main NAT function is `nf_nat_fn()` is called by the following hooks: `NF_INET_PRE_ROUTING`, `NF_INET_POST_ROUTING`, `NF_INET_LOCAL_OUT`, and `NF_INET_LOCAL_IN` hooks. The NAT module is called at all points in the network stack where packets enter or leave the host. The hooks are registered with a priority such that the CT module is always called before the NAT module, and the packet filtering module is always called after the NAT module. This is necessary because the NAT module depends on the states generated by the CT module.

The `nf_nat_fn()` function starts by obtaining a CT state for the packet being processed. If a CT state is not found it means that the CT module was unable to track the packet and thus it cannot be translated by NAT either. If a CT state is found and the state is `IP_CT_NEW`, the NAT rule for the packet is obtained. If no NAT rule is found, the function returns `NF_ACCEPT` without altering the packet. If a NAT rule for the packet exists, the function `nf_nat_packet()` is called. This function calls `manip_pkt()` which in turns calls the protocol-specific function by the same name as defined by a `struct nf_nat_protocol`. If the `manip_pkt()` fails to alter the packet according to the NAT rule, the packet is dropped. The return value of the protocol-specific `manip_pkt()` determines the final verdict on the packet by the NAT module.

The manipulation of the tuples generated by the CT module is performed by the function `nf_nat_setup_info()`. This function is called when a packet belonging to a new connection is sent or received. The setup function calls the `get_unique_tuple()` function which in turn calls the protocol-specific function defined in the `struct nf_nat_protocol` of the protocol. In the case of UDP, the function obtains an external IP address and port number. The tuples in the CT state are then updated with the new external IP address and port number. Due to the changes in the tuples, the hash value of the tuples will no longer point to the correct entry in the CT module's hash table. Therefore the hash value is recalculated and the CT state is moved accordingly.

5. POTENTIAL VULNERABILITIES

The design and implementation of the connection tracking and NAT modules contains several features that can make a device running NAT vulnerable to denial-of-service attacks if not configured correctly. Hash tables are inherently vulnerable to hash collision attacks. If an attacker sends a large number of packets with different source ports, the hash of

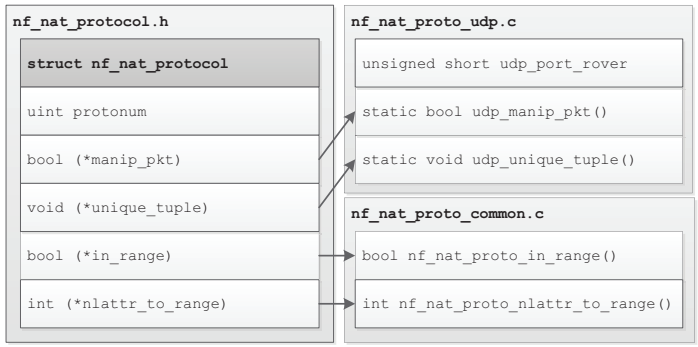

Figure 7: `struct nf_nat_protocol` and its initializations for the UDP protocol.

the tuples used by the CT module will collide depending on the size of the hash table. In Linux the default number of hash buckets depends on the amount of RAM available. For instance, a system with 4 GB of RAM has 16384 hash buckets and supports a maximum of 65536 simultaneous connections. If the maximum number of connections is reached the hash table will contain an average of four entries per hash bucket. Such a system will not become unstable if flooded by single source, but routers typically do not have 4 GB of RAM.

Another issue with NAT, is that packets are dropped if the external address space is exhausted. Flooding a gateway with packets from different source port numbers, will force NAT to allocate an external address to each flow. Because it only takes a single packet to reserve an external address, the external address space can be depleted very quickly. This problem is tied to the length of protocol timeout values. The default timeout value for unidirectional UDP traffic is 30 seconds and it is possible to send 65535 packets with all of the available source port numbers within this timeout period. A shorter timeout period results in external addresses being freed more quickly and thus denial of service is less likely. The timeout should not be set lower than the typical RTT of a connection in order to avoid replies to legitimate traffic from being dropped.

A third issues with NAT, is that connection tracking may requires transport-layer information or even application-layer information. In order to inspect data at these layers, fragmented IP packets must be defragmented and this is problematic. The defragmentation process requires the NAT device to buffer incoming packets until the whole IP packet can be assembled. The buffering consumer system resources and depending on the number of fragmented packets, this may be a problem. Because of this, some NAT devices refuse to process fragmented IP packets. Like the hash-collision attack, it may be possible to increase memory usage in a NAT device by sending many fragmented packets.

Obvious solutions to hash-collision and address exhaustion attacks are to allocate more memory to the hash table and reduce timeout values for connections. Another solution could be to dynamically adjust hash table sizes and timeout values to mitigate attacks. Dynamic adjustment of timeout values would provide better service when a device is not heavily loaded. With respect to the fragmentation issue, the simplest solution is to drop fragmented IP packets.

6. CONCLUSIONS

Netfilter is a complex and important part of the Linux kernel. This paper has given a detailed overview of the core functions and data structures used by the connection tracking and NAT modules. Some issues in connection tracking and NAT in general were also discussed. Hopefully this paper will serve as a future reference for developers who wish to contribute to the development of Linux and Netfilter.

7. REFERENCES

[1] J. Engelhardt and N. Bouliane. Writing netfilter modules, July 2012. `http://inai.de/documents/Netfilter_Modules.pdf`.

[2] IANA. Address family numbers. October 2012. `http://www.iana.org/assignments/address-family-numbers/address-family-numbers.xml`.

[3] IANA. Assigned internet protocol numbers. October 2012. `http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml`.

[4] K. Wehrle, F. Pahlke, H. Ritter, D. Muller, and M. Bechler. The linux networking architecture: Design and implementation of network protocols in the linux kernel. August 2004. ISBN 978-0131777200.

Mobile IPv6 in Linux Kernel and User Space

Jouni Korhonen*
Mutkatie 2 A 4
1100 Riihimäki
Finland
jouni.korhonen@iki.fi

ABSTRACT

Linux implementation of Mobile IPv6 is divided into both kernel space and user space. The modifications done in kernel are minimal and mainly concentrate around IPv6 extension header processing. The actual logic and operation is entirely realized in user space daemon. This paper describes briefly how Mobile IPv6 is implemented in Linux.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*IPv6, Mobile IPv6*; C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*Internet*; D.4.4 [**Operating Systems**]: Communications Management—*Network communication*

Keywords

Mobile IPv6, Linux Kernel, XFRM6

1. INTRODUCTION

Mobile IPv6 (MIPv6) [14] is an extension to IPv6 [3], which allows a mobile node (MN) to change the point of attachment to the Internet while maintaining a stable IP address. This *stable* IP address is called the Home Address (HoA). A topological anchor for the HoA is a Home Agent (HA) gateway/router, and the link where the HoA topologically belongs to is called a home link. While the mobile node is away from its home link, the mobile node registers its current IPv6 address, the Care-of Address (CoA) with the Home Agent using the binding management signalling. The mobile node and the home agent use IP tunnelling between each other and thus allow tunnelling IP traffic sourced from or destined to HoA via the home agent with other Correspondent Nodes (CN) in Internet. The tunnel is between the mobile node's current location, the CoA, and the Home Agent Address (HAA). Mobile IPv6 also supports Route Optimization (RO), where a mobile node and a correspondent node exchange IPv6 traffic directly between each other bypassing the home agent. The Ro requires additional return routability procedure (RRP), after which the mobile node and the correspondent node with the help of the home agent are able to establish a good enough level of security between each other. Figure 1 shows the basic Mobile IPv6 architecture and related messaging paths between a mobile node, a home agent and a correspondent node.

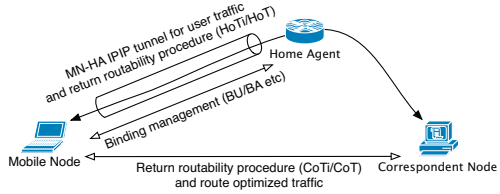

Figure 1: Simplified Mobile IPv6 architecture

According to the standard [14] the mobile node and the home agent must use transport mode IPsec [9] Encapsulating Security Payload (ESP) [8] with non-NULL payload authentication algorithm to provide data origin authentication, connectionless integrity, and optional anti-replay protection for all Mobile IPv6 signalling between the mobile node and the home agent. This is to prevent malicious hosts to redirect traffic and cause denial of service attacks towards victim hosts. The required security association (SA) can be manually configured or using Internet Key Exchange (IKE) [4] protocol. The security between the mobile node and the correspondent node is realized using a Binding Authorization Data option in relevant signalling messages. The option uses and *Kbm* key, which is established during the return rout ability procedure.

This brief paper goes through Mobile IPv6 implementation in Linux (*MIPL*) kernel version 3.5.3. Mobile IPv6 version implemented in Linux kernel is based on the older version of the protocol [7] with small cherry picked error corrections from the current protocol version [14]. Mobile IPv6 use of IPsec for securing the traffic between the mobile node and the home agent that is implemented in Linux kernel is also based on the older version of the security protocol framework [1]. The main difference between the current IPsec for

*J. Korhonen participated to the course as a lazy independent contributor and wrote this during the last night.

OS Protocols '12 Espoo, Finland

Mobile IPv6 specification [4] and the older one is the version of IKE.

The Linux implementation of Mobile IPv6 is divided into both kernel space and user space. The kernel side support is kept minimal and consists of a single loadable module (`mip6.ko`) and a number of small patches mainly for handling of the new IPv6 extension headers required by Mobile IPv6. Linux Mobile IPv6 implementation makes extensive use of the XFRM framework [19, 18, 10]. Most of the logic and the actual Mobile IPv6 operation is implemented entirely in user space, including the required extensions to IPv6 neighbor discovery protocol (NDP) [11]. The original MIPL code is hosted at *http://www.mobile-ipv6.org/* but also available as part of the Proxy Mobile IPv6 [6] code branch at *http://www.openairinterface.org/openairinterface-proxy-mobile-ipv6-oai-pmipv6*. The most recent branch of Linux Mobile IPv6 is *UMIP*, see *http://www.umip.org*. This paper emphasises on pinpointing the Mobile IPv6 originated changes in Linux kernel and most essential concepts implemented in user space. We surface the XFRM framework but do not really dive into the details of it.

The rest of the paper is divided as follows. Section 2 describes what IPv6 extension and mechanisms Mobile IPv6 involves, and how they are implemented in Linux. In Section 3 we look briefly into packet processing, tunnel management and other Mobile IPv6 specific operations. Finally we conclude the paper in Section 4.

2. MOBILE IPV6 IMPLEMENTATION

2.1 Implementation strategy

As mentioned earlier, Linux Mobile IPv6 implementation is divided into both kernel space and user space parts. This design decision allows for extending Mobile IPv6 rather easily as long as no new IPv6 extension headers are required. Proxy Mobile IPv6 a good example of this. At the same time the changes in kernel can be kept minimum, which typically means more robust implementation.

The user space part is referred in this paper usually as MIPL. The kernel code takes care of Mobile IPv6 related IPv6 extension header handling (for both incoming and outgoing IPv6 packets), tunnelling traffic between an mobile node and a home agent, and XFRM-based IPsec solution for Mobile IPv6 [10]. An user space Mobile IPv6 *daemon* then has to take care of the rest of Mobile IPv6 logic, tunnel & binding management, IPsec policies & security associations management for Mobile IPv6 purposes, ICMPv6 [2] and IPv6 neighbor discovery protocol [11, 17] extensions. It should be noted that the Mobile IPv6 coed used for this paper did not include IKE support for managing security associations. IPsec part of the configuration is read from a configuration file that the user space Mobile IPv6 daemon then uses to set up possible security associations for protecting traffic between a mobile node and a home agent.

The main Mobile IPv6 support code is located in *mip6.c* file, which can be compiled as a loadable module. We will discuss the details of this file later in Section 3.1. Unless specifically noted, all kernel source files are located under *$SRC/net/ipv6/* directory. Possible user space MIPL source files are located under *$MIPL/src/* directory, unless noted differently. The same code base for the user space and the kernel space apply to all Mobile IPv6 functions: mobile nodes, home agents and correspondent nodes.

It is quite surprising but there are no any major data structure within kernel that would be Mobile IPv6 specific. Maybe the XFRM policy database and the security association database could be considered one but those are, at the end, just normal XFRM data structures. On the other hand, the user space Mobile IPv6 daemon then has multiple vital data structures. The most important mobile node side data structure is the list of `struct bulentry` entries (see the *bul.h* file) used by an mobile node for maintaining its bindings to home agents and correspondent nodes. Likewise both home agent and correspondent node maintain a list of `struct bcentury` (see the *bcache.h* file) entries for their mobile node bindings.

2.2 Mobility header

Mobile IPv6 adds a new extension header *Mobility Header* (MH), which is used to convey Mobile IPv6 specific messages such as Binding Update (BU), Binding Acknowledgement (BA) , Home Test (HoT) & Home Test Init (HoTI), and Care-of Test Init (CoTi) & Care-of Test (CoT). The mobility header has a Next Header value 135. The mobility header is used by all mobile nodes, home agents and correspondent nodes for their Mobile IPv6 signalling. Refer Section 6.1.1 of [14] for the mobility header format.

An user space program can send and receive mobility headers using RAW sockets. The RAW socket filtering and local delivery support kernel implementations for the `IPPROTO_MH` can be found from the kernel *raw.c* and *mip6.c* files. In user space the MIPL the *mh.c* file and functions `mh_init()`, `mh_recv()` and `mh_send()` serve as good examples of using RAW sockets for the mobility header handling.

2.3 IPv6 extension headers

Mobile IPv6 introduces two new IPv6 extension headers: *Home Address Option* (HAO) carried in an IPv6 Destination Option and *Type 2 Routing Header* (RH2) carried in an IPv6 Routing Header. The home address option is sent by a mobile node (when away from its home link) to carry its HoA to the recipient. Refer Section 6.3 of [14] for the home address option format.

The type 2 routing header is used by an home agent and a correspondent node to send traffic directly to mobile node's care-of address using source routing (while the mobile node is away from its home link). While the packet is on transit the type 2 routing header contains the home address of the mobile node. Refer Section 6.4 of [14] for the format of the type 2 routing header.

The extension header processing is mainly located in the kernel *exthdrs.c* file for incoming packets. The home address option is currently the only implemented destination option (see the `ipv6_dest_hao()` function), which swaps the IPv6 header source address with the home address found in the home address option. The type 2 routing header handling is added into the `ipv6_rthdr_rcv()` function, which swaps the IPv6 header destination address with the home address found in the type 2 routing header. Note that the Mobile IPv6 extension header handling takes only place if there is a matching `xfrm_state` and `xfrm_policy` established for incoming traffic that may contain a type 2 routing header or a home address destination option. A packet with a type 2 routing header or a home address destination option that arrives out of blue is dropped as a security measure. These extension header handler functions are called by the

`ip6_input_finish()` function located in *ip6_input.c* file using the normal `inet6_protos[]` handler chain.

The extension header handling and processing is possible for an user space program using `sendmsg()` and `recvmsg()` socket functions and their *ancillary data* structures. Of course using RAW sockets is another possibility to manipulate extension headers. Good examples can be found from the MIPL *mh.c* file and `mh_recv()` and `mh_send()` functions.

2.4 Neighbor discovery protocol

The IPv6 neighbor discovery protocol [11] has been extended to carry the home agent information. A new '`H`'-bit has been added into the Router Advertisement (RA) message header flags field and a corresponding '`R`'-flag into the Prefix Information Option (PIO) for the home agent IPv6 address (note, it is an address, not a prefix). These changes have **no** kernel changes. An user space application has to implement its own router advertisement daemon to advertise on the home link that this specific node is a home agent.

The MIPL home agent implementation (see the *ha.c* file) merely monitors for incoming router advertisements to learn other home agents around it for Dynamic Home Agent Discovery (DHAAD) purposes. On the other hand, the home agent has to defend for the home address on the home link while the mobile node is away and also perform Duplicate Address Detection (DAD) [17] for every configure home address. For this purpose the MIPL implements a subset of the neighbor discovery protocol. The home agent initiated address resolution and the DAD are implemented in the *ndisc.c* file. The mobility header also implements a very brief router advertisement handler and the Neighbor Unreachability Detection (NUD) as part of its movement detection algorithm (see the *movement.c* file and e.g., the `md_start()` function as a starting point).

2.5 Extensions to ICMPv6

Mobile IPv6 adds several new ICMPv6 [2] types. These include the Home Agent Address Discovery Request Message (type 144), the Home Agent Address Discovery Reply Message (type 145), the Mobile Prefix Solicitation Message Format (type 146) and the Mobile Prefix Advertisement Message Format (type 147). These ICMPv6 messages have **no** kernel impact and in MIPL their implementations can be found from the *icmp6.c* file. See also the *dhaad.c* and *mpdisc_*.c* files for further ICMPv6 handling details.

The Mobile IPv6 user space daemon uses RAW sockets for implementing required new ICMPv6 extension. This concerns both neighbor discovery protocol and other Mobile IPv6 specific ICMPv6 types.

2.6 Security and use of XFRM

The XFRM framework for Linux and its IPv6 extension XFRM6 [19, 18] is a rather large and complex subsystem. The XFRM uses stackable destination architecture for efficient outbound packet processing. It is basically a link list of `dst_entry{}` structures that are constructed once the packet is outputted and its destination gets looked up. The generic address family agnostic XFRM framework is located in *$SRC/net/xfrm/* and its IPv6 extensions in the usual place *$SRC/net/ipv6/*. Mobile IPv6 makes extensive use of the XFRM for three purposes:

- Protecting binding management traffic between the mobile node and the home agent using IPsec in transport mode and/or protecting the user traffic using IPsec in tunnel mode. The use of IPsec optional.
- Inserting the home address option into IPv6 packets with a mobility header between the mobile node and the home agent, and into any IPv6 packet targeted to correspondent nodes in route optimization mode.
- Inserting the type 2 routing option into IPv6 packets destined to a home agent or correspondent nodes in route optimization mode.

The XFRM implementation for Mobile IPv6 is described in [10] in greater detail. In general the `xfrm_state{}` structure corresponds to an IPsec security association (i.e., IPsec SA), the `xfrm_policy{}` corresponds to the IPsec security policy (i.e., IPsec SPD) and the `xfrm_tmpl{}` is a glue between the policy and the state. Mobile IPv6 adds two type handlers to the XFRM: one for the IPv6 home address option handling and one for the IPv6 type 2 routing header handling. There are also two new XFRM modes for Mobile IPv6 route optimization purposes (see the *xfrm6_mode_ro.c* file) i.e., to add space for the new IPv6 extension headers. The mode can be invoked using the `XFRM_IN_TRIGGER` or `XFRM_MODE_ROUTEOPTIMIZATION` mode specifier when creating a new XFRM template. The latter is used for inserting the appropriate extension header for a matched IP flow and the former is used for triggering the user space daemon to initiate the route optimisation procedure.

There is one peculiar aspect in the binding management signalling originated by either the home agent or the correspondent node. The home agent or the correspondent node user space daemon has to insert the type 2 routing header manually into the binding management messages it sends to the mobile node (using `sendmsg()` and its *ancillary data* (see the `mh_send()` function in the MIPL *mh.c* file). Before sending a packet with a mobility header the home agent or the correspondent node makes sure no XFRM policy matches to the packet sent to a mobile node. Figure 2 summarises how XFRM is used along with Mobile IPv6 signalling and route optimization. The figure is taken from [10].

The user space part of the Mobile IPv6 is entirely responsible for managing the XFRM policies and states. The MIPL *xfrm.c* file contains a number of utility functions to manipulate the XFRM policies and states. The communication between the user space daemon and the kernel uses `netlink` [15]. The required `netlink` utility functions are located in the MIPL *rtnl.c* file, which then uses the *libnetlink.c* located in the *$MIPL/lib/libnetlink/* directory.

Within the kernel, the `struct xfrm_state` and its member `coaddr` carries the care-of address for the established XFRM state. When a XFRM state gets created for either the destination header (`IPPROTO_DSTOPT`) or the routing header (`IPPROTO_ROUTING`), the care-of address must be defined. Also, the new XFRM handlers for Mobile IPv6 IPv6 extension header handling carry the `XFRM_TYPE_LOCAL_COADDR` or `XFRM_TYPE_REMOTE_COADDR` bit in the respective `xfrm_type` structure `flags` field. This information is used when the XFRM framework searches for a policy that is specific for Mobile IPv6 purposes.

2.7 Address selection

Every IPv6 host should implement a default address selection algorithm [5, 16], which is used e.g., to select the

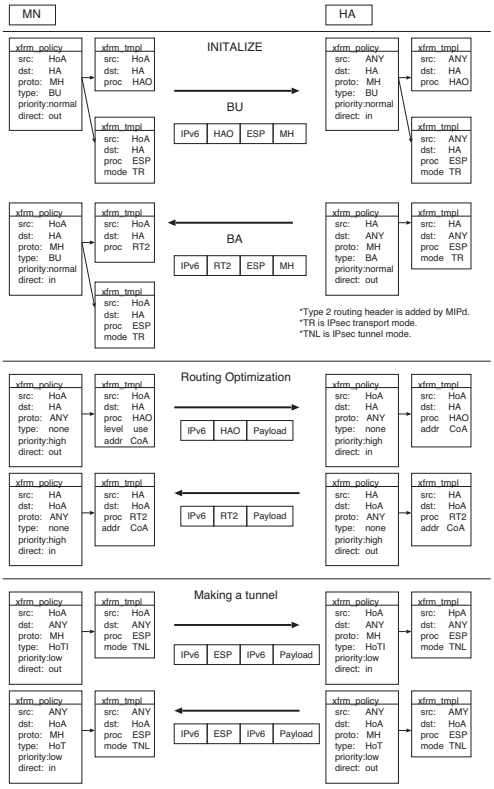

Figure 2: XFRM usage example with Mobile IPv6 (taken from [10])

preferable source address for a given destination address. Mobile IPv6 extends the default address selection. Specifically the source address selection `Rule 4` (see Sections 5 and 6 of [**?**]) allows preferring home address over other addresses (assuming the mobile node has multiple IPv6 addresses of the same scope). The `ipv6_get_saddr_eval()` function in the kernel *addrconf.c* file is the place where the rule gets verified. An user space program can, however, affect the address selection policy using `setsockopt()` and manipulating the `IPV6_PREFER_SRC_COA` flag in the IPv6 address preferences flags [12]. Setting this flag basically downplays the `Rule 4` and the kernel may pick up other source address than the home address.

2.8 Tunnelling

Finally, the last part of the Mobile IPv6 implementation we look into is the IPv6 traffic tunnelling between the mobile node and the home agent is realised. Apart from the normal IP-in-IP tunnelling support [13] basically any Linux kernel, there are `no` kernel changes. The user space Mobile IPv6 daemon is responsible for: a) creating a tunnel interface and b) maintaining the tunnel endpoint addresses based on the current binding state. MIPL uses a sequence of `ioctl()` commands to create/modify/delete a tunnel interface. See functions `tunnel_add()`, `tunnel_del()` and `tunnel_mod()` in the MIPL *tunnelctl.c* file. The tunnel is established between an mobile node and a home agent when the mobile node is away from the home link. The tunnel is create both mobile node and home agent after a successful exchange of Binding Update and Binding Acknowledgement messages. The mobile node side of the tunnel is bound to the current care-of address and the home agent side of the tunnel is bound to the home agent address (see `tunnel_mod()` function in the *tunnelctl.c* file).

In order to route specific traffic into the tunnel the MIPL uses `netlink` interface to add/delete/modify routes to the kernel. See functions `route_add/del/mod()` in the MIPL *rtnl.c* file. When the mobile node is away from the home link, it points the default route to the tunnel. When the mobile node moves between networks and the care-of address also changes, the mobile node has to modify the tunnel addresses again.

Then, when traffic gets tunnelled over the IPIP tunnel and when it gets route optimized? When a mobile node and a home agent set up a binding state, they always implicitly setup XFRM state that route optimizes traffic between the two i.e., traffic is exchanged using type 2 route header and home address destination option. However, when a packet sources by the mobile node is destined to some other node than the home agent, it gets reverse tunnelled. Similarly when a packet arrives to the home agent from the internet that is destined to the mobile node, the packet is tunnelled to the mobile node. However, whenever the mobile node sees reverse tunnelled traffic with other nodes than the home agent, it attempts to initiate the route optimization procedure (assuming route optimization was enabled in the first place). In order this to happen the mobile node has to first install appropriate XFRM policies for triggering route optimization.

3. MOBILE IPV6 OPERATION

This section gives a short overview how packet processing works for incoming and outgoing packets. We also look what happens when Mobile IPv6 gets initialised. All these aspects are only looked from the kernel point of view. Last we have few words actual experimentation.

3.1 Initializing Mobile IPv6

When the Mobile IPv6 starts up i.e., the *mip6.ko* kernel module is loaded using typical Linux means and the `__init_mip6_init()` gets called in the *mip6.c* file, the kernel does two things. First, the `mip6_mh_filter()` handler function for mobility header filtering gets installed for RAW sockets using the `rawv6_mh_filter_register()` function. After this user space programs may filter IPv6 packets with mobility header using RAW sockets.

Second, two XFRM type handlers get registered into the XFRM framework using the `xfrm_register_type()` function. These handlers allow e.g., input and output procession of the IPv6 home address destination option and the type 2 routing header. We already discussed for what these XFRM types and later corresponding policies and rules are used for in Section 2.6.

After these three handler registrations the Linux Kernel is ready for Mobile IPv6 operation. The rest is then left

for the user space programs to set up. Initially there are no XFRM policies or states for any flows so the kernel will drop all incoming packets with the type 2 routing header and no packet gets inserted with either the home address option or the type 2 routing header.

3.2 Input packet processing

When an IPv6 packet arrives to an it gets eventually into the `ip6_input_finish()` function. If a RAW socket has been opened with a mobility header filter, the IPv6 packet gets delivered into the user space by `rawv6_local_deliver()` function. If there are required XFRM policies and templates available for type 2 routing header (see Figure 2), then eventually `ipproto->handler(skb)` processes the extension header and once successfully completed passed the packet through XFRM handlers into the upper layer application. These both functions are located in the *ip6_input.c* file. Note that the default XFRM handlers do nothing for the type 2 routing header in the input chain (see `mip6_rthdr_input()` function in the *mip6.c* file).

Figure 3 shows an overview how Mobile IPv6 related packet flows through the IPv6 input handlers and invokes the XFRM, and eventually reaches the upper layers.

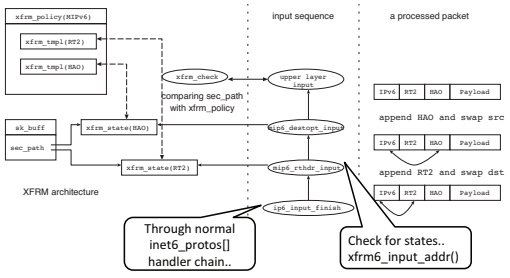

Figure 3: Input packet processing example with Mobile IPv6 (modified from [10])

3.3 Output packet processing

For output processing we look into two cases: first when the packet contains a mobility header and second for the packet without mobility header.

When an IPv6 packet with a mobility header is sent from an application, it eventually after completing the `dst_output` (passed by one of the output functions in the *ip6_output.c* file) gets caught by XFRM handlers. We assume there is a matching XFRM policy and template for the IPv6 packet address pair and the mobility header installed. The XFRM handler function `mip6_dstopt_output()` in the *mip6.c* file inserts the home address option into the IPv6 packet, swaps the source address in the IPv6 header and in the home address option, and passes the packet further in the XFRM chain. Eventually the packet gets sent out (see Figure 2).

The operation for an IPv6 packet without the mobility header is similar but now depending on the installed XFRM policy and template either the home address option or the type 2 routing header gets inserted into the IPv6 packet. These extension header insertions are done either in the `mip6_destopt_output()` or the `mip6_rthdr_output()` functions in the *mip6.c* file. Note that the user space application has to insert the type 2 routing header manually into the IPv6 packet when the packet contains the mobility header. In this case the kernel does not do it.

Figure 4 shows how Mobile IPv6 related packets flow through the IP stack and make use of the stackable destination feature of the XFRM before reaching the final IP output.

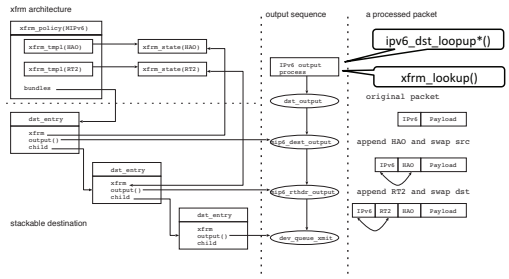

Figure 4: Output packet processing example with Mobile IPv6 (modified from [10])

4. CONCLUSION

This paper made a quick survey how Mobile IPv6 is implemented in Linux. The implementation in split into a small set of kernel space functions and user space daemon. The kernel mainly takes care of new extension header processing and packet forwarding, while the real Mobile IPv6 logic is in the user space daemon. Linux Mobile IPv6 makes also extensive use of the XFRM framework for both securing Mobile IPv6 using IPsec and also processing both IPv6 home address destination option and type 2 routing header under specific cases. In addition to few lines long patches here and there the main Mobile IPv6 kernel implementation is in two files *exthdrs.c* and *mip6.c* with both being relatively short ones.

The kernel implementation is rather straight forward excluding the XFRM framework part. The real complexity, after all, is found in the user space daemon that handles the Mobile IPv6 protocol logic. The division between the kernel and user space is nice in a sense that it makes further development of the Mobile IPv6 simple as long as no new IPv6 extension headers need to be included or the XFRM framework can be used for IPsec and other packet mangling purposes. For example, there is a Proxy Mobile IPv6 implementation build on top of the Linux MIPL implementation with minimal changes to the user space application.

5. REFERENCES

[1] J. Arkko, V. Devarapalli, and F. Dupont. Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents. RFC 3776, Internet Engineering Task Force, June 2004.

[2] A. Conta, S. Deering, and M. Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443, Internet Engineering Task Force, Mar. 2006.

[3] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, Internet Engineering Task Force, Dec. 1998.

[4] V. Devarapalli and F. Dupont. Mobile IPv6 Operation with IKEv2 and the Revised IPsec Architecture. RFC 4877, Internet Engineering Task Force, Apr. 2007.

[5] R. Draves. Default Address Selection for Internet Protocol version 6 (IPv6). RFC 3484, Internet Engineering Task Force, Feb. 2003.

[6] S. Gundavelli, K. Leung, V. Devarapalli, K. Chowdhury, and B. Patil. Proxy Mobile IPv6. RFC 5213, Internet Engineering Task Force, Aug. 2008.

[7] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. RFC 3775, Internet Engineering Task Force, June 2004.

[8] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP). RFC 2406, Internet Engineering Task Force, Nov. 1998.

[9] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, Internet Engineering Task Force, Nov. 1998.

[10] K. Miyazawa and M. Nakamura. IPv6 IPsec and Mobile IPv6 implementation of Linux. *Proceedings of Linux Symposium*, 2:85–94, July 2004.

[11] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861, Internet Engineering Task Force, Sept. 2007.

[12] E. Nordmark, S. Chakrabarti, and J. Laganier. IPv6 Socket API for Source Address Selection. RFC 5014, Internet Engineering Task Force, Sept. 2007.

[13] C. Perkins. IP Encapsulation within IP. RFC 2003, Internet Engineering Task Force, Oct. 1996.

[14] C. Perkins, D. Johnson, and J. Arkko. Mobility Support in IPv6. RFC 6275, Internet Engineering Task Force, July 2011.

[15] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol. RFC 3549, Internet Engineering Task Force, July 2003.

[16] D. Thaler, R. Draves, A. Matsumoto, and T. Chown. Default Address Selection for Internet Protocol Version 6 (IPv6). RFC 6724, Internet Engineering Task Force, Sept. 2012.

[17] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862, Internet Engineering Task Force, Sept. 2007.

[18] H. Yoshifuji, K. Miyazawa, M. Nakamura, Y. Sekiya, H. Esaki, and J. Murai. Linux IPv6 Stack Implementation Based on Serialized Data State Processing. *IEICE Trans Commun (Inst Electron Inf Commun Eng)*, E87-B(3):429–436, March 2004.

[19] H. Yoshifuji, K. Miyazawa, Y. Sekiya, H. Esaki, and J. Murai. IPv6 IPsec and Mobile IPv6 implementation of Linux. *Proceedings of Linux Symposium*, 1:507–523, July 2003.

Device Agnostic Network Interface

Jonne Soininen
Aalto University
P.O. Box 11000
FI-00076 AALTO
jonne.soininen@renesasmobile.com

ABSTRACT

There is a lack of up-to-date, easily approachable documentation of the Linux kernel networking stack. This paper tries to address that gap by describing parts of the stack - the device agnostic network interface (net_device), and the kernel network buffers (sk_buff). In addition, the paper describes the two different approaches for using the device agnostic network interface - the traditional and the New API - seeking to describe the strengths of the New API, and prove them, at least partly, through an experiment.

Keywords

Linux, kernel, networking, drivers

1. INTRODUCTION

This paper concentrates on describing how network device drivers are connected to the Linux kernel, and how the packets are transferred between the device driver and the Linux kernel. The aim of this paper is to give an overview to an interested reader of that interface, and its anatomy. The focus is on this interface itself, and the buffer management related to it. Other topics, such as the actual device drivers, and the networking protocols, such as Internet Protocol version 4 (IPv4)[1], and Internet Protocol version 6 (IPv6)[3] are described in other papers of the Network Protocols in Operating Systems course. Thus, we only mention them to describe the functionality of the device agnostic network interface.

This article is based on the Linux kernel version 3.5.4 as published at *kernel.org*[4]. There is no other special reason for picking up this specific kernel version than it was the newest at the time this article was started. The references to the Linux kernel files are made as relative file paths to the specific file or directory in the Linux kernel source tree. For example, *include/net/ping.h* refers to a file called ping.h in the directory include/net under the kernel source tree. Absolute paths, e.g. */var/log* are used to refer to a directory or a file in a Linux distribution file system. The used path names should be generic. However, there are sometimes variations between different Linux distributions. This article is based on the Debian Project distribution[5].

Network drivers are connected to the Linux kernel differently from other drivers. As generally in Unix, and also in Linux, all device drivers are usually represented by a file in the */dev* directory (or in Linux case also through files in the */sys* directory), and the user space applications use these to control and communicate with the device. However, network devices do not have file nodes associated to them in the same way. Therefore, there is no associated */dev* file for the Ethernet driver. Network devices are connected to Linux via network interfaces. The device interface between the network protocols and the device drivers is the *device agnostic interface*, and the actual data is transferred in the network buffer structure called `sk_buff`.

2. DEVICE INTERFACE OVERVIEW

The device agnostic interface, also known as the Net Device Interface, is the interface between the device specific device driver and the network protocol stack. For instance, between the Ethernet network card device driver, and the Internet Protocol version 6. As the name already indicates, the same interface is used regardless of the physical device, and the network protocol stack used. There is a good reason to have such a generic interface. It allows the protocol stack not to be changed when a new device driver is introduced to the system. In addition, the device drivers do not have to be tweaked if a new network protocol is introduced. The need for this becomes even more evident when looking at the hundreds of network device drivers at the *drivers/net*. You can find in that location drivers for many different network devices ranging from ham radio, to the modern ultra high speed Ethernet devices.

In a considerably simplified view, the life cycle of a networking device can be categorized in the following stages.

1. Initialization
2. Opening a network interface
3. Transmitting packets (sending and receiving packets)
4. Closing the interface
5. Removal

Clearly, the receiving or sending of packets usually do happen multiple times and very much in any order during the

lifetime of a device. In some setups, for instance in a consumer's computer, the system receives considerably more packets than it ever sends. On the other hand, in other setups, such as a web server, the system actually sends many more packets than it ever receives. In addition, in a router configuration, the device practically sends and receives an equal amount of traffic. Thus, generally speaking, both receiving and transmitting packets are performance sensitive operations.

In the following, we will look at these different steps in more detail, and explain how they work, and what happens in each step, and how the steps are implemented in the Linux kernel. We will start with opening and closing a network interface. In the end of the section, we explain the packet transmission itself.

2.1 Network driver initialization and removal

The first step in the life of a network device is the initialization. This is the process where the Linux kernel is introduced the new device. A device is initialized, for example, at the loading of the module to the kernel. The device initialization comprises of the following steps.

1. *Allocate the network device memory*:

   (`alloc_netdev`) The initialization sequence starts with the memory allocation for the structure holding the device information - `net_device`. We will examine this more in detail when looking at the `net_device` structure in this paper.

2. *Perform optional setup checks*:

   (`netdev_boot_setup_check`) Checks boot time options of the device. This is an optional step. In addition, this step is used by legacy device drivers. It is not in use in modern device drivers.

3. *Setup the device*:

   In this step, the `net_device` structure is populated with the device specific information, and the device specific routines.

4. *Registering the device to kernel*:

   (`register_netdev`) Registers the device to the kernel, and informs the kernel the device is ready to be used.

The device removal is considerably more straight forward. To remove the device from kernel, the device has to be only unregistered (`unregister_netdev`), cleaned up, and ultimately freed (`free_netdev`). This usually happens at the removing of the device module from the kernel.

2.2 Starting and stopping a network interface

Before sending, and receiving packets, the network interface has to be opened. Opening the network interface may happen either during the boot or at runtime. For example, in modern computers the inbuilt interfaces (such as the Ethernet card) can be set up already during the boot sequence, if just the physical media (i.e. the Ethernet cable) is present. However, there are also other interfaces that are mainly set up during runtime. These interfaces include, for instance, cellular network interfaces.

During runtime, the network interface can be turned up using such command line tools as `ifconfig`. An example of turning on the first ethernet interface (*eth0*) can be seen in the following:

```
root@Debian:~$ ifconfig eth0 up
```

In the device agnostic interface, setting the interface up results to a call to the function `dev_open()`, which allocates the transmit and receive resources, registers the interrupt handler to the OS, sets the device watchdog, and informs the kernel the device is up. The watchdog timer is used to detect network interface halts, and if the watchdog is activated the device is reset.

The opposite can be performed by typing the following to the command line.

```
root@Debian:~$ ifconfig eth0 down
```

This triggers the call for the `dev_close()`, which does the exact opposite of the `dev_open()`: The interrupt handler is deregistered, the transmit and receive resources are freed, and the device is set to be down.

2.3 Packet transmission

Once the network interface is up, packets can start to flow over the interface to both directions. We will try to give an overview how sending and receiving packets is done in Linux in this section. We will start with receiving packets, and then look at sending packets.

2.3.1 Sending packets

When sending packets from the Linux host towards the network, the packets are usually queued in the transmission queue. The Linux network queue policy and internals are outside of the focus of this paper. Eventually, the packets will need to be transmitted, and `dev_hard_start_xmit()` is called to perform the packet transmission. As the result of this, the `dev_hard_start_xmit()` will call the `start_xmit()`, which points to the device specific transmission method. The device specific transmission method then moves the packets to be sent to the device itself. Figure 1 shows the flow graphically.

2.3.2 Receiving packets

The receiving of packets is a slightly more complex and sending packets, but not much. The packets can arrive at any point, and the device has to have a method to tell the kernel that a packet has arrived. This is done by the software interrupt registered at the network interface open. Two different strategies to handle the device interrupts exist. Thus, also two different kernel internal APIs exist: The traditional interface, and the New API (NAPI)[7]. The former create a software interrupt every time a packet is received. The NAPI, however, only creates a software interrupt for the first packet in the burst, and then schedules the kernel to poll the interface. This strategy can save considerably in software interrupts, and therefore is well adapted for high

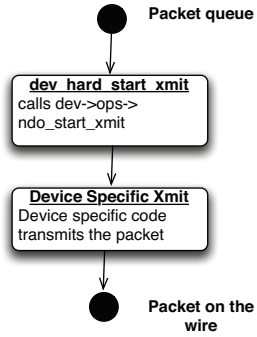

Figure 1: Transmitting packets to the network interface

speed interfaces. We will discuss the traditional interface versus the NAPI later in this paper. As the traditional interface is a bit simpler, we will go through its functionality in this section.

You can find the receive packet flow in Figure 4. When a packet comes into the device the device raises an interrupt. This interrupt calls the device specific software interrupt routine. The software interrupt routine copies the packet into memory, and calls `netif_rx()` routine, which provides the packet to the Linux kernel.

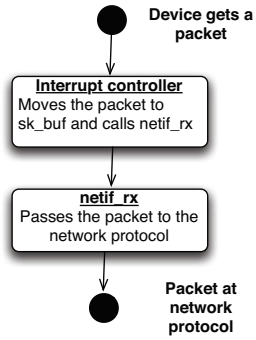

Figure 2: Receiving packets in the network interface

2.4 Related data structures, and functions

This section will look at the data structures and functions, which implement the device agnostic interface. In the end of this section, we will also look at the work split between the device agnostic interface, and the device specific code.

2.4.1 Data structures

Table 1 shows the most important data structures used in the device agnostic interface. The main structure is the `net_device` structure. This structure is the definition of the actual interface. The structure itself is very complex, and has many fields. It makes little sense for us to go through all the fields in this paper. An interested reader can look up the

Data Structure	Location	Description
net_device	`include/linux/netdevice.h`	The device agnostic interface definition.
net_device_ops	`include/linux/netdevice.h`	The device operations.
net_device_stats	`include/linux/netdevice.h`	Device statistics.

Table 1: Device agnostic interface relevant data structures

Field	Description
char name	Name of the interface.
unsigned long mem_start, mem_end	Shared memory area.
unsigned long base_addr	device I/O address.
unsigned int irq	Device interrupt request number.
*const struct net_device_ops *netdev_ops*	Structure containing the device operations.
*const struct header_ops *header_ops*	Hardware header description.
unsigned int flags	Interface flags.
unsigned char perm_addr	Permanent hardware address.
unsigned char addr_len	Hardware address length.

Table 2: A sample of the `net_device` structure fields.

structure in the header file. However, the most interesting fields with the explanation are listed in the following.

The `net_device` structure contains the device name, the I/O specific fields, the device status, the interface flags, the device hardware address, and the device statistics among other information. The structure also contains the pointer to the `net_device_ops` structure specific to this device. Table 2 shows a sample set of the `net_device` structure fields. The structure itself is very complicated, and an interested reader should look directly at the source code.

The `net_device_ops` structure lists the pointers to the device operations functions. We will look at the operations in next Section.

2.4.2 Functions

The support functions for the device agnostic interface are implemented in *net/core/dev.c*. These include the following functions: `alloc_netdev()`, `netdev_boot_setup_check()`, `register_netdev()`, `unregister_netdev()`, `free_netdev()`, `dev_open()`, `dev_close()`, `dev_hard_start_xmit()`, and in addtion `netif_rx()` we have mentioned before. These include the generic implementation of those functions, which then call the device specific implementations, which are linked as pointers in the `net_device_ops`.

2.4.3 Work split between the device agnostic interface and the device specific implementation

The device agnostic network interface is the generic interface to the kernel for the device drivers. Therefore, it should minimally include any device specific information. This is also partly achieved, though some at least link-layer specific information is embedded in the device agnostic interface as well. On the other hand, the device driver - the device

Field	Description
*sk_buff *next, *prev*	Support for packet listing and queuing.
*sock *sk*	Socket owning the packet.
*net_device *dev*	The device the packet is belonging (incoming or outgoing).
*unsigned char *head*	Pointer to the start of the packet.
*unsigned char *data*	Pointer to the data of the packet.
sk_buff_data_t tail	Pointer to the end of the data.
sk_buff_data_t end	Pointer to the end of the packet.
sk_buff_data_t transport_header	Pointer to the beginning of the transport protocol header.
sk_buff_data_t network_header	Pointer to the beginning of the network protocols header.
sk_buff_data_t mac_header	Pointer to the beginning of the Layer 2 header.

Table 3: Relevant fields in `sk_buff` structure

specific implementation should contain the functions, and data structures specific for the device, and the device driver should provide the translation from the device specific implementation to the generic.

As described above, the device agnostic interface is defined in the `net_device` structure in *include/linux/netdevice.h*. The functions needed for the device initialization, opening, transmission, closing, and de-initialization can be found at *net/core/dev.c*. These definitions and functions are common for all the device drivers.

The device specific implementations can be found at *drivers/net*. When a new device is initialized the device specific initialization sets up the `net_device` structure with the information needed by the device agnostic interface. This includes also the callback functions to operate the device. The pointers to these functions are set to the `net_device_ops` structure. The device specific implementation is outside the scope of this paper.

3. LINUX NETWORK BUFFER MANAGEMENT

An important part of the network device functionality is the network buffer management. This includes the management of the incoming packets, and the out-going packets in their buffers. This part includes both the packet buffer structure itself, and the functions used to manipulate the packet buffers.

3.1 `sk_buff` data structure

The Linux kernel uses a data structure called `sk_buff` to store the packets. The `sk_buff` structure is defined in *include/linux/skbuff.h*.

The `sk_buff` is a complex structure. Though, it is not quite as complex as the `net_device` structure, it is complex enough not to be listed completely here. An interested reader can look at the header file to find the additional information. The relevant information in the focus of this paper is listed in Table 3.

The most important fields in the focus of this paper are `head`, `data`, `tail`, `end`, `transport_header`, `network_header`, and `mac_header`. These tell us where the different parts

Function	Description
alloc_skb()/dev_alloc_skb()	Allocates a `sk_buff`.
skb_clone()	Clones the `sk_buff` structure without copying the payload data.
build_skb()	Builds a buffer
kfree_skb()/dev_kfree_skb()	Frees the `sk_buff`

Table 4: Basic operations on `sk_buff`

of a packet are. The `head` points to the beginning of the whole packet. The `data` points to the beginning of the data of the packet. What is the "data" depends on the protocol level currently processed by the kernel. When processing the network protocol (IPv4 or IPv6) the `data` points to the beginning of the network protocol header. However, on higher layer - on transport layer (such as TCP[2]), the `data` points to the transport layer protocol. Thus, the `data` pointer changes as the packet moves up the protocol stack. The `tail` behaves similarly pointing into the end of the data. The `transport_header`, `transport_header`, `network_header`, and `mac_header` point to their respective header locations in the packet buffer. Figure 3 shows how these different pointers relate to the buffered packet.

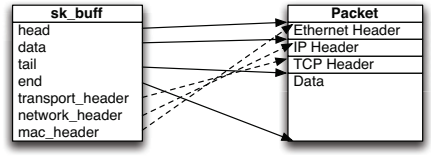

Figure 3: Relationship between the `sk_buff` fields and the buffered packet

3.2 Managing `sk_buff` structures

The *net/core/skbuff.c* contains a great variety of operators to manage `sk_buff`s. These operations range from queue management to the basic operations on `sk_buff`s. In the interest of space, we will here only focus on the basic operations used to manage the Linux kernel network buffers. These very basic operations are listed in Table 4.

The `alloc_skb()` and `dev_alloc_skb()` allocate an `sk_buff`. The later is used from the device, and the previous in the kernel. The `kfree_skb` and `dev_kfree_skb` are used to free an allocated buffer. The `skb_clone` provides copying the `sk_buff` structure. However, it does not copy the buffer contents, but just the structure. This allows manipulating the structure, for instance, on different protocols layers without the overhead of copying the packet memory. The `skb_put()` provides a mechanism to put data into the buffer.

4. NAPI

The New API (NAPI) interface was introduced to the Linux kernel to improve the network performance. The aims of the NAPI are listed in the following.

Interrupt mitigation means minimization of the software interrupts the device creates when a new packet arrives. The traditional interface creates a software interrupt at arrival of every packet. This causes both

considerable and unnecessary load on the system. When performing high speed networking the system may become overwhelmed by processing just the software interrupts. The NAPI addresses this consideration.

Packet throttling means dropping the packet as soon as possible when the system cannot handle the inflow of packets. With NAPI, the packets can be dropped already at the device without even introducing them to the kernel if the system is not able to process the packets.

The NAPI was introduced in the Linux kernel version 2.6, and by the kernel version we are concentrating on here - 3.5.4 - many, if not most, of the device drivers have moved to use the NAPI. However, a device driver can be still written to use the traditional interface, and the traditional interface is still supported by the kernel today. This may introduce considerable constraints on the devices performance, though. We will be looking at the performance considerations more in detail in Section 5.

In this section, we will look at how the NAPI works.

4.1 Overview

The basic principle of NAPI is to move the processing of the packets away from the interrupt handler to the kernel as much as possible. This does not mean that the software interrupt handler would not be used at all anymore, however. Let's remind ourselves first how the traditional interface works.

The traditional interface registers a software interrupt handler, which is called at packet arrival. The handler then calls `netif_rx()` to provide the received packet to the Linux kernel. The NAPI changes this behavior. Instead of calling `netif_rx()` a NAPI interrupt handler disables the software interrupt, and schedules the *polling* of the interface. The flow in the interrupt handler is on high level as follows.

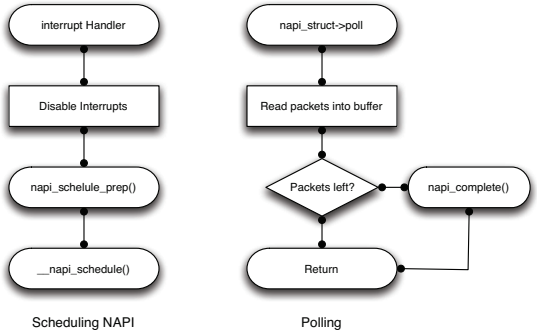

Figure 4: NAPI flow - left side shows the first call on interrupt handler, and right side the kernel poll sequence

1. Disable the packet reception interruption for the device.
2. Check if it is possible to schedule the incoming packet polling with `napi_schedule_prep`.
3. Schedule the polling of the device with `__napi_schedule`.

Practically, this means the packet processing is removed from the interrupt handler and postponed to a time in the future. In addition, the interrupt masking means that no new interrupts from this device are coming in before the interrupts are enabled, again.

To process the packets, the kernel needs to *poll* the device to pull the packets into the kernel. The scheduling of the poll tells the kernel that there are packets to be polled. In a suitable time for the kernel, the kernel will perform the poll operation. Currently, the Linux kernel supports multiple poll queues per device, which can all be polled separately. However, here we simplify the actual functionality a bit. The poll queues are stored in the `napi_struct` structure, which includes the *poll* function among other items. The *poll* function is device specific, and transfers all the packets waiting for processing to the kernel. The NAPI uses `netif_receive_skb()` instead of `netif_rx`. When the job is complete, the poll function sets the interrupts on again, and performs `napi_complete`. This turns the device back to the interrupt driven mode. This disables the polling of the device, if there is nothing to process anymore.

4.2 NAPI requirements for devices

The NAPI introduces certain requirements for the device. The requirements are listed in the following.

- Direct Memory Access (DMA) ring support, or enough memory to store the packets in the software device.
- Ability to turn the related interrupts off.

Devices that can support these requirements can be supported by NAPI. Otherwise, the device has to use the traditional interface. However, devices not compatible with the NAPI requirements will unlikely be high speed networking devices, and thus, the traditional interface should be adequate for them.

5. PERFORMANCE CONSIDERATIONS

In this Section, we examine the performance differences between NAPI and the traditional device interface. First we will examine the results from available literature, and then we will explain the test system created to perform own testing of the comparison.

5.1 Literature study

A study on NAPI was described in a USENIX article in 2001[6]. The results are also published on the Linux Foundation[8] web page on NAPI[7]. The USENIX article describes a situation where a Linux system used a router collapses after a certain number of packets per second after the Central Processing Unit (CPU) load goes to 100%. The main reason is the number of interrupts the traditional interfaces causes on the CPU during when the computer receives a high speed data transfer. The high number of interrupts is caused by

the traditional interface causing an interrupt at every received packet. [1]

The performance improvement of NAPI is based on suppression of interrupts especially when receiving packets. On traditional interface the input packet ratio to the interrupts is 1. The system creates an interrupt at every packet that is received. NAPI can improve this considerably while there are packets in the input queue. The Linux Foundation page refers a ratio of 17 interrupts per 890k packets/second. Trying to push this amount of interrupts per second through any system will cause a considerable load, and most probably choke the system. In a situation like this, the NAPI clearly brings benefits.

However, if the input buffer cannot be kept occupied throughout the transmission, the benefits of NAPI start to decline. At worst, the NAPI reduces again to packet per interrupt performance with the additional overhead of NAPI. Though, the NAPI overhead is not considered to be meaningful, the benefits can only be obtained from a constantly occupied input buffer. Though, it may seem that this is not a significant problem as the packets are coming at a slow enough speed for the CPU to handle them, it can cause the CPU load to raise, and reduce the capacity of the computer to perform other duties.

5.2 Testing NAPI against the traditional interface

Due to the lack of up to date literature, an extremely small scale test was conducted. The purpose of the test was to try to measure the differences between NAPI to the traditional interface. In this section, we will describe first the test setup, and then analyze the results of the tests.

5.2.1 Description of the test setup

In the absence of suitable hardware, the test was performed on using Linux virtual machines. The test setup is described in Figure 5. The test setup consists of two Linux virtual machines running a Apple MacBook Pro running Mac OS X. The virtualization software used was VirtualBox from Oracle[9]. Iperf[10] was used to test the network speed.

One of the machines was setup as an Iperf client. This machines was running Debian 6.0 with the Linux kernel version 3.5.4 described in this paper. The other machine was setup as an Iperf server. The two virtual machines were connected by using the VirtualBox's This machine's performance was measured as it was the machine receiving the transmission. The network interface selected was VirtualBox's emulated Intel Gigabit Ethernet device. The kernel version (2.6.26) was selected as it was the last version of the network card device driver (e1000), which had a possibility to turn off NAPI. Test runs were run both on NAPI supported, and non-NAPI supported modules for comparison purposes. The number of packets received, and the number of interrupts handled were recorded before and after the test run, and the difference was calculated - hence, calculating the number of packets received, and number of interrupts handled during the session.

[1]The Salim paper was written in 2001. Hence, there is little point to repeat the packets per second numbers, which caused problems to Linux at that time. It seems that little up to date literature exists currently.

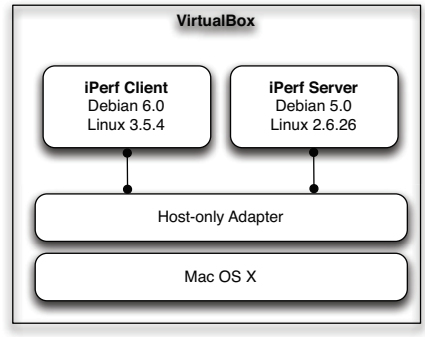

Figure 5: Test setup with two Linux virtual machine in VirtualBox on Mac OS X host

Two versions of `e1000.ko` were compiled - one with NAPI turned on, and one with NAPI turned off. The `e1000.ko` default parameters were used. Three different tests were conducted - transferring 300 MB, and 1 GB of data over Transmission Control Protocol (TCP)**??** using Message Segment Size 1 KB, and 300 MB over User Datagram Protocol (UDP)**??** in 150 byte size packets. The reason for selecting these packet sizes was in TCP to control the packet size, and in UDP to try the performance with small packets.

This was not done with real hardware, we did not study the VirtualBox's architecture, and limitations, and the measuring of the packets, and the interrupts was done by hand at looking at Linux's counters. Thus, this test was by no means scientifically representative. The only target of this test was to empirically see an indication of the difference, and, of course, personal learning. Obviously, the reader should take the results with a pinch of salt. In this setup, it was not possible to isolate all other variables, and the setup itself created certainly effects on the results. In addition, one should perform many more test runs to create a representative data set.

5.2.2 Analysis of the test results

The result of the test runs can be found in Table 5. The most difference can be found in the number of raised interrupts. The traditional API 300MB TCP data transfer causes almost and the 1 GB transfer over ten times as many interrupts than the NAPI transfer. The reader should notice that the number of interrupts per packet even in traditional API is much less than the number of packets transferred. This is because the `e1000` driver does tries to throttle the number of interrupts even in traditional API case.

The differences in data transfer speed are more moderate - in the 300 MB transfer the difference is ca 14%, and in the 1 GB transfer ca 8%. As stated in the previous section, these results are not exactly trustworthy due to the lack of adequate testing, and the lack of real hardware. However, they indicate that using NAPI can make the data transfer

Protocol	Method	Packets	Interrupts	Size	Speed (MB/s)
TCP	Traditional	318399	58472	300MB	23.4
TCP	NAPI	318399	6102	300MB	27.2
UDP (150 bytes)	Traditional	2097103	1428427	300MB	0.76
UDP (150 bytes)	NAPI	2097163	1438770	300MB	0.75
TCP	Traditional	1086789	263699	1GB	20.60
TCP	NAPI	1086789	19280	1GB	22.5

Table 5: Comparison between NAPI and traditional interface test results

considerably more efficient.

The UDP results actually describe a different situation. The traditional API was actually a bit quicker. However, for some reason the NAPI interface seems to have transferred more packets. The reasons for this were not studied.

6. SUMMARY

This paper described the functionality of the device agnostic network interface, the network buffer management, and the New API (NAPI).

The device agnostic network interface is defined in *include/linux/netdevice.h*. It defines the data structure (`net_device`) that stores the device information, and the includes the functions to operate the interface. The functions are implemented in *net/core/dev.c*.

The network buffers in Linux are stored in the `sk_buff` data structure. The `sk_buff` is defined in *include/linux/skbuff.h*. Linux provides a set of tools to manage the `sk_buff`s. These operations are implemented in *net/core/skbuff.c*.

We also looked at the NAPI, which increases network performance by performing interrupt mitigation, and packet throttling at the device. We also discussed the performance aspects of NAPI, and concluded the NAPI performs better than the traditional interface, when the input buffers can be kept occupied, and the ratio between interrupts per packets can be kept low. In the worst case, the performance degrades to the level of the traditional API at 1 interrupt/packet.

7. REFERENCES

[1] Postel J., "Internet Protocol", IETF, RFC 791, September 1981.

[2] Postel J., "Transmission Control Protocol", IETF, RFC 793, September, 1981.

[3] Deering S. and Hinden R., "Internet Protocol, Version 6 (IPv6) Specification", IETF, RFC 2460, December 1998.

[4] kernel.org, "The Linux Kernel Archives", kernel.org, 2012.

[5] Debian Project, "Debian", www.debian.org, 2012.

[6] Salim J., Olsson R., Kuznetsov A., "Beyond Softnet", USENIX, 2001.

[7] The Linux Foundation, "napi", http://www.linuxfoundation.org/collaborate/workgroups/networking/napi, 2009.

[8] The Linux Foundation, "The Linux Foundation", http://www.linuxfoundation.org/, 2012.

[9] Oracle, "Oracle VM VirtualBox", https://www.virtualbox.org/, 2012.

[10] Sourceforge, "Iperf", http://sourceforge.net/projects/iperf/, 2012.

[11] Postel J., "Transmission Control Protocol", IETF, RFC 793, September 1981.

[12] Postel J., "User Datagram Protocol", IETF, RFC 768, August 1980.

Network device drivers in Linux

Aapo Kalliola
Aalto University School of Science
Otakaari 1
Espoo, Finland
aapo.kalliola@aalto.fi

ABSTRACT

In this paper we analyze the interfaces, functionality and implementation of network device drivers in Linux. We observe that current real-life network drivers are very heavily optimized compared to a minimum baseline implementation. Additionally we analyze some of the optimized functionality in greater detail to gain insight into state-of-the-art network device driver implementations.

Keywords

device driver, network interface, linux

1. INTRODUCTION

Devices in Linux are divided into three main categories: block, character and network devices. Block devices provide adressable and persistent access to hardware: after writing to a specific block address subsequent reads from the same address return the same content that was written. In contrast, character devices provide unstructured access to hardware. Character device may be a separate devices such as a mouse or a modem or part of the internal computer architecture, for instance a communications bus controller.

Character devices do not typically do buffering while block devices are accessed through a buffer cache. In some cases block devices can also have character device drivers specifically for bypassing the cache for raw I/O.

Network devices in Linux differ significantly from both block and character devices [4]. Data is typically sent by an upper program layer by placing it first in a socket buffer and then initiating the packet transmission with a separate function call. Packet reception may be interrupt-driven, polling-driven or even alternate between these modes depending on configuration and traffic profile. An optimized driver wants to avoid swamping the CPU with interrupts on heavy traffic but the driver also wants to maintain low latency in low traffic scenarios. Therefore it cannot be categorically stated that interrupt-driven would be better than polling or vice versa.

In this paper we will examine network device drivers in Linux starting from driver basics and interfaces and finishing in a partial analysis of certain functionality in the e1000 network driver. The device agnostic network interface is described in another report [9], so it will not be discussed in detail here.

2. OVERVIEW

2.1 Network drivers and interfaces

A network driver is in principle a simple thing: its purpose is to provide the kernel with the means to transmit and receive data packets to and from network using a physical network interface. The driver is loaded into the kernel as a *network module*.

The network device driver code interfaces to two directions: the Linux network stack on one side and device hardware on the other. This is illustrated in Figure 1. While the network stack interface is hardware-independent the interface to network interface hardware is very much dependent on the hardware implementation. The functionality of the network interface is accessed through specific memory addresses, and is thus very much non-generic. For the purposes of this paper we try to remain generic where possible and go into device-dependent discussion only where absolutely necessary.

In Linux kernel source tree network driver code for manufacturers is located in *linux/drivers/net/ethernet/* [7]. Our later analysis of Intel e1000 driver code will be based on the code that is in *linux/drivers/net/ethernet/intel/e1000/*.

2.2 Common data structures

While in this paper we focus on the driver internals, two data structures common with the device agnostic network interface require mentioning at this point: *net_device* and *sk_buff*.

A network interface is described in Linux as a *struct net_device* item. This structure is defined in *<linux/netdevice.h>*. The structure includes information about the interface hardware address, interface status, statistics and many other things.

Packets are handled in transmission as socket buffer structures (*struct sk_buff*) defined in *<linux/skbuff.h>*. In addi-

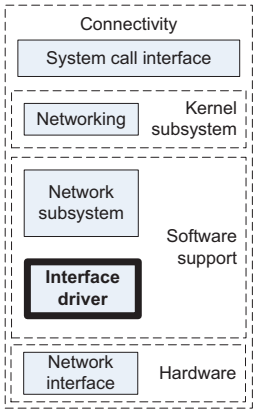

Figure 1: Device driver role in network connectivity

tion to containing pointers to actual packet data it contains (among many things) information about the owner socket, owner device and also possible requirements for protocol offloading in hardware.

3. DRIVERS IN PRACTISE

In this section we look into network device drivers in the real world. First we look into the minimal functionality a network driver needs to have to transmit and receive packets. Then we look at the e1000 driver for Intel network interface cards and summarize its code organization. Finally we look to into the optimizations and other code that the e1000 driver has in addition to the bare minimum functionality.

3.1 Simple driver

A minimal network device driver needs to be able to transmit and receive packets. In order to be able to do this the driver needs to do a series of tasks: the physical device needs to be detected, enabled and initialized [6]. After initialization the device can be readied for transmitting and receiving packets. For all this to be possible we need to have an understanding of the network device, bus-independent device access, PCI configuration space and the network device chipset's transmission and receive mechanisms.

For device detection it is possible to use the *pci_find_device* function with our device manufacturer's vendor ID and device ID. This gives us a pointer to the device. The device can then be enabled with *pci_enable_device* using the pointer as the only argument.

The purpose of bus-independent device access is to provide an abstraction for performing I/O operations independent of bus type. Most commonly this is used by drivers in reading and writing to memory-mapped registers on a device. One practical example of this woud be the cleaning of TX/RX buffers, in which 0 is written to the relevant registers.

As previously discussed, network interfaces are seen by the Linux networking stack as devices in *net_device*. This structure is then updated with actual device values during device initialization. In initialization the driver retrieves the base memory address for the device registers, which is needed in all subsequent addressing of device registers.

The transmission and receiving mechanisms on hardware are device-dependent and can be discovered either through manufacturer documentation such as [8] (on which this simple driver is based) or by reverse engineering. Usual areas of interest are the transmit and receive memory buffers. These buffers can be implemented on hardware as, for instance, ring buffers. A ring buffer is a data structure of fixed size in which the end of the buffer is connected to the beginning. Thus the buffer appears circular.

The ring buffer does not contain the whole of packet data, but rather a set of descriptors. These descriptors point to addresses in the system main memory. When a packet is ready for transmission the device reads the packet content from the location pointed to by the descriptor, uses direct memory access (DMA) to place the data into its own FIFO and then transmits the data from the physical network port.

On the RX side the memory is also a ring buffer with the difference that it will overwrite oldest stored packet once full. On reception an interrupt is generated resulting in an interrupt handler getting called, which will then process the received packet. Later in interrupt throttling feature analysis we will observe a more complicated process.

3.2 e1000

In this section we observe some points of interest in the e1000 code.

3.2.1 struct e1000_adapter

The structure *struct e1000_adapter* in *e1000.h* is the board-specific private data structure. It contains many interesting fields for features related to VLANs, link speed and duplex management, packet counters, interrupt throttle rate, TX and RX buffers, and also the operating system defined structures *net_device* and *pci_dev*. An adapter may have multiple RX and TX queues active at the same time.

3.2.2 e1000_probe

Function *e1000_probe* initializes an adapter. This includes operating system initialization, configuration of the adapter private structure *struct e1000_adapter* and a hardware reset. In addition the function does hardware integrity checks and Wake-on-LAN initialization.

3.2.3 e1000_open

When an e1000 network interface is activated by the system the function *e1000_open* is called. This function allocates resources for transmit and receive operations, registers the interrupt (IRQ) handler with the operating system, starts a watchdog task and notifies the stack that the interface is ready. Since e1000 driver uses the New API (NAPI) [2] it also calls *napi_enable* before enabling the adapter IRQ.

3.2.4 Buffers

e1000 has receive (RX) and transmit (TX) buffers implemented as ring buffers with definitions in *e1000.h*.

The RX (*e1000_rx_ring*) and TX (*e1000_tx_ring*) buffer sizes can be defined when loading the driver module with possible buffer sizes between 80 and 4096 descriptors each. While increasing buffer sizes may help in avoiding packet drops during heavy CPU utilization large buffers may cause issues with increased latency and jitter.

3.3 Simple driver vs. e1000 driver

It is obvious that the e1000 driver is hugely more complex than the simple driver: whereas the simple driver can be implemented in under 1000 lines of code (LOC) the e1000 driver has closer to 20.000 LOC. The difference in complexity comes mainly from additional features and optimization of basic functionality such as interrupt rate control on packet reception.

e1000	simple	name	description
✓	✓		TX and RX buffers
✓	-		Device statistics
✓	-	*active_vlans*	Virtual LAN
✓	-	*wol*	Wake-on-LAN
✓	-	*tx_itr, ...*	Interrupt throttling
✓	-		Link speed control
✓	-		HW checksumming
✓	-		NAPI
✓	-		Thread safety
✓	-		Watchdog functions

Table 1: Driver private structure comparison

One comparison that can be done to get some insight about the feature differences between the simple driver and the e1000 driver is to look into their private data structures and see what fields they contain. These names are shown, if applicable, and grouped into categories with descriptions in Table 1.

4. FEATURES

In this section we look into specific feature implementations in the e1000 driver, namely interrupt throttling and TCP offloading.

4.1 Interrupt throttling

When the network interface receives a series of packet it can send an interrupt, buffer the packet and wait for a poll later on, or buffer the packet and send an interrupt when the allocated buffer size hits a given threshold or a time limit is reached. This set of different behaviours is an area that has plenty of opportunities for optimization for different traffic profiles. Polling is typically less resource-consuming with heavy traffic, whereas interrupt-driven operation is better for low latencies when the traffic is not very heavy.

Without throttling the basic interrupt processing proceeds as shown in the following list. If NAPI is enabled, the interrupt handler *e1000_intr* will in practise schedule the adapter for NAPI polling.

1. Network card transmits or receives a packet.
2. Network card generates an interrupt.
3. CPU handles the interrupt. This includes handling state information and executing an interrupt handler.
4. Device driver takes actions based on the interrupt cause.
5. CPU resumes its previous activity.

The e1000 driver has an internal interrupt throttle rate (ITR) control mechanism that has the option to dynamically adjust the interrupt frequency depending on the current traffic pattern. In effect this adds a new step in the interrupt processing between steps 1. and 2. of the previous listing. In this new step the network card delayes generating the interrupt in order to transmit or receive more packets during the delay.

The interrupt throttling functionality is controlled using the *InterruptThrottleRate* parameter in *e1000_param.c*. The purpose is to set the maximum number of interrupts per second the interface generates when receiving or transmitting packets [1]. Similar to other configurable parameters, the ITR can be set on loading the driver module, for instance *"modprobe e1000 InterruptThrottleRate=1"* [3].

The ITR value has different meanings depending on what value it is set to. This is presented in the following list. The default setting is 3.

- 0 = off
- 1 = dynamic
- 3 = dynamic conservative
- 4 = simplified balancing
- 100..100000 = max interrupts/sec fixed to this value

The dynamic options adjust the actual ITR by categorising the current traffic periodically into one of three different classes:

- *bulk latency* for large number of packets/sec of normal size,
- *low latency* for small number packets/sec or a significant portion of small packets,
- *lowest latency* for very small number of packets/sec or almost completely small packets.

In *dynamic* mode the driver reduces latency quite aggressively for small packets: ITR is increased all the way to 70000 if the traffic profile is lowest latency. For bulk traffic ITR value is 4000. Low latency profile results in an ITR of 20000. The *dynamic conservative* mode is identical to dynamic mode with the exception that lowest latency traffic is capped to the same ITR value of 20000 as the low latency traffic.

In *simplified* mode the maximum ITR is set based on TX/RX traffic ratio. With equal traffic in both direction ITR is set to 2000 , with completely asymmetric traffic ITR is set to 8000. Cases falling between these extremes result in an ITR somewhere between 2000 and 8000. *Off* mode turns off all interrupt moderation resulting in one interrupt per packet.

As an interesting sidenote the driver takes a shortcut for link speeds less than 1Gbps: in this case ITR is fixed to 4000.

Within e1000main.c the functions relevant to ITR are:

- *e1000_update_itr*

- *e1000_set_itr*

Current traffic profile is calculated in *e1000_update_itr* based on number the of packets and bytes accumulated since previous interrupt. This results in a very up-to-date view of current traffic, though the calculation must be quite minimal to avoid heavy overhead. The calculation has three different base cases, one for each current traffic profile. The calculation is shown in Figure 2.

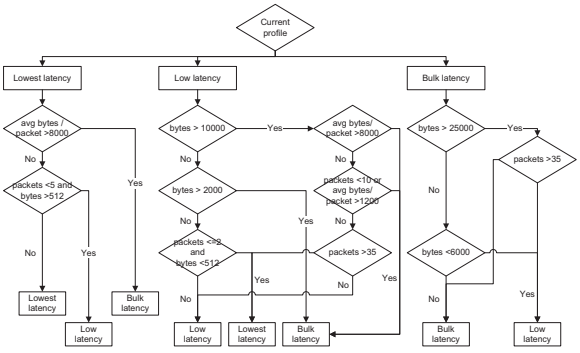

Figure 2: Per-interrupt traffic class determination

It is noteworthy that in no condition does the traffic class update from bulk to lowest with a single ITR update: it must go through the low latency class first.

After the traffic class has been updated *e1000_set_itr* then sets the actual ITR depending in ITR configuration and the traffic class. Finally the actual throttling of interrupts is achieved using the throttle value and several absolute and packet timers in implemented in the device controller hardware.

4.2 Protocol offloading

Protocol offloading in general is meant to lessen the load of the system CPU while maintaining high throughput by doing some protocol processing already on the network interface hardware. It has been studied in detail with various scenarios, for instance by Freimuth et al [5].

TCP Segmentation offloading (TSO) is the functionality that does the cutting of packets to maximum transmission unit (MTU) sized chunks on transmission. This size is typically 1500 bytes. A TSO-capable network card also needs to support TCP checksum offloading as checksum calculation is an important part of cutting the oversized data packet into valid smaller chunks.

Since the operating system must know if the network interface supports TSO the support is defined by the e1000 driver in the hardware features of the *net_device* structure. Once the networking stack above driver level is aware that the network card and driver support TSO it is possible to deliver large chunks of data to the network card. Typical packet size from the upper stack point of view is 64kB.

The actual TSO operation is done in function *e1000_tso*. In the beginning the function checks that the *sk_buff* is meant to be segmentation offloaded using function *skb_is_gso*. After this the IP header and TCP checksum creation is handled separately for IPv4 and IPv6, after which the remaining critical fields are filled and the newly cut packet is placed into the TX ring buffer.

TCP and UDP checksum offloading is enabled for packet reception in *e1000_configure_rx*. Actual RX checksum verification is done in *e1000_rx_checksum*. From the driver point of view the checksum checking is simple: the driver simply checks for various possible issues with the checksum and finally, if the packet is a TCP or UDP packet and the hardware checksum indication is ok, increment the checksum ok counters.

TX checksums are unsurprisingly done in *e1000_tx_csum*. The TX checksumming appears to be only for TCP, not for UDP.

5. CONCLUSION

Modern network device driver and hardware handles functionality that has originally been the responsibility of the operating system and the CPU, such as packet fragmentation and checksum computation. This leads to significant complexity in the driver compared to a more feature-limited approach.

One of the interesting findings in this paper was that the e1000 driver has significant control over the packet processing mode: it is capable of adjusting its interrupt generation depending on current traffic profile. This also places a significant amount of burden on the network interface hardware and drivers as a weak implementation might well hamper the performance of the whole system.

Overall our analysis of a minimal network device driver and the state-of-the-art e1000 driver shows that the current real-life drivers are thoroughly optimized and feature-rich.

6. REFERENCES

[1] Interrupt Moderation Using Intel GbE Controllers. `http://www.intel.com/content/dam/doc/application-note/gbe-controllers-interrupt-moderation-appl-note.pdf`, April 2007.

[2] NAPI. `http://www.linuxfoundation.org/collaborate/workgroups/networking/napi`, November 2009.

[3] Linux e1000 base driver overview and installation. `http://www.intel.com/support/network/adapter/pro100/sb/cs-032516.htm`, November 2011.

[4] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, Third Edition*. 2005.

[5] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey. Server network scalability and TCP offload. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[6] M. L. Jangir. Writing network device drivers for linux. *Linux Gazette*, (156), November 2008.

[7] lxr@linux.no. http://lxr.linux.no/\#linux+v3.6.7/drivers/net/ethernet, November 2012.

[8] Realtek. *RTL8139(A/B) Programming guide: (V0.1)*. 1999.

[9] J. Soininen. Device agnostic network interface, 2012.

Anatomy of a Linux bridge

Nuutti Varis
Aalto University School of Electrical Engineering,
Department of Communications and Networking
P.O.Box 13000, 00076 Aalto, Finland
Email: {firstname.lastname}@aalto.fi

ABSTRACT

Ethernet is the prevalent Local Area Networking (LAN) technology, offering a cost efficient way to connect end-hosts to each other. Local area networks are built by networking devices called switches, that forward Ethernet frames between end-hosts in the network. The GNU/Linux operating system can be used to create a software based switch, called a *bridge*. This paper explores the architecture, design, and implementation of the Linux bridging component, and attempts to chart some of the processing characteristics of the frame forwarding operation, inside the bridge and in the operating system as a whole.

1. INTRODUCTION

Network devices, called switches (or synonymously, *bridges*) are responsible for connecting several network links to each other, creating a local area network. Conceptually, the major components of a network switch are a set of network ports, a control plane, a forwarding plane, and a MAC learning database. The set of ports are used to forward traffic between other switches and end-hosts in the network. The control plane of a switch is typically used to run the Spanning Tree Protocol (STP) [15], that calculates a minimum spanning tree for the local area network, preventing physical loops from crashing the network. The forwarding plane is responsible for processing input frames from the network ports, and making a forwarding decision on which network ports the input frame is forwarded to.

Finally, the MAC learning database is used to keep track of the host locations in the LAN. It typically contains an entry for each host MAC address that traverses the switch, and the input port where the frame was received. The forwarding decision is based on this information. For each unicast destination MAC address, the switch looks up the output port in the MAC database. If an entry is found, the frame is forwarded through the port further into the network. If an entry is not found, the frame is instead flooded from all other network ports in the switch, except the port where the frame was received. This latter provision is required to guarantee the "plug-and-play" nature of Ethernet.

In addition to Linux, several other operating systems also implement local area network bridging in the network stack. FreeBSD has a similar bridging implementation to Linux kernel, however the FreeBSD implementation also implements the Rapid Spanning Tree Protocol (RSTP). The FreeBSD bridge implementation also supports more advanced features, such as port MAC address limits, and SNMP monitoring of the bridge state. OpenSolaris also implements a bridging subsystem [12] that supports STP, RSTP, or a next generation bridging protocol called TRILL [14].

There has been relatively little evolution in bridging since the inception of the STP. Switches have evolved in conjunction with other local area network technologies such as Virtual LANs [16], while the STP has been incrementally extended to support these new technologies. Currently, there are two practical next-generation solutions for switching: RBridges (TRILL), and the Shortest Path Bridging (SPB) [1]. Both TRILL and SPB diverge from STP based bridging in several important ways. Some of the key differences are improved loop safety, more efficient unicast forwarding, and improved multicast forwarding. Additionally, the well known scalability issues [2] of the local area networks, and the advent of data center networking has also created a number of academic research papers, such as SPAIN [10], Port Land [11], VL2 [6], DCell [7], and BCube [8].

This paper explores the architecture, design and the implementation of the Linux bridging module. In addition, the paper also analyzes the processing characteristics of the Linux bridging module by profiling the kernel during forwarding, and observing various counters that track the performance of the processors and the memory in the multi-core CPU. The design and implementation of STP in the Linux bridge module is considered out of scope for this paper.

The rest of the paper is structured as follows. Section 2 presents an overview of the central data structures of the Linux bridge, creation of a Linux bridge instance, and the processing flow of an incoming frame. Next, Section 3 describes the forwarding database functionality of the bridge implementation. Section 4 describes the experimentation setup, and analyzes some of the performance related aspects of the bridging module and the operating system. Finally, Section 5 finishes the paper with some general remarks of local area networks and the Linux bridging implementation.

2. OVERVIEW

The architectural overview of the Linux bridging module is divided into three parts. First, the key data structures for the bridging module are described in detail. Next, the configuration interface of the Linux bridging module is discussed by looking at the bridge creation and port addition mechanisms. Finally, the input/output processing flow of

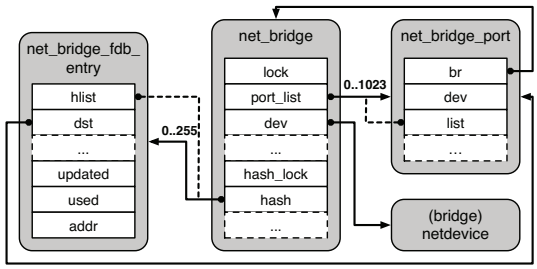

Figure 1: Primary Linux bridge data structures

the Linux bridging module is discussed in detail.

2.1 Data structures

The Linux bridge module has three key data structures that provide the central functionality for the bridge operation. Figure 1 presents an overview of the most important fields and their associations in the three key data structures. The main data structure for each bridge in the operating system is the `net_bridge`. It holds all of the bridge-wide configuration information, a doubly-linked list of bridge ports (`net_bridge_port` objects) in the field `port_list`, a pointer to the bridge netdevice in the field `dev`, and the forwarding database in the field `hash`. The technical details and the functionality of the hash array table are described in 3.1. Finally, the field `lock` is used by the bridge to synchronize configuration changes, such as port additions, removals, or changing the various bridge-specific parameters.

Each bridge port has a separate data structure `net_bridge_port`, that contains the bridge port specific parameters. The field `br` has a back reference to the bridge that the port belongs to. Next, the `dev` field holds the actual network interface that the bridge port uses to receive and transmit frames. Finally, position of the data structure object in the `net_bridge->port_list` linked list is stored in the field `list`. There are also various configuration parameter fields for the port, as well as the port-specific state and timers for the STP and IGMP [5] snooping features. IGMP snooping will be detailed in Section 3.2.

Finally, the third key data structure for the Linux bridge module is the `net_bridge_fdb_entry` object that represents a single forwarding table entry. A forwarding table entry consists of a MAC address of the host (in the field `addr`), and the port where the MAC address was last seen (in the field `dst`). The data structure also contains a field (`hlist`) that points back to the position of the object in a hash table array element in `net_bridge->hash`. In addition, there are two fields, `updated` and `used`, that are used for timekeeping. The former specifies the last time when the host was seen by this bridge, and the latter specifies the last time when the object was used in a forwarding decision. The `updated` field is used to delete entries from the forwarding database, when the maximum inactivity timeout value for the bridge is reached, i.e., $current\_time - updated > bridge\_hold\_time$.

2.2 Configuration subsystem

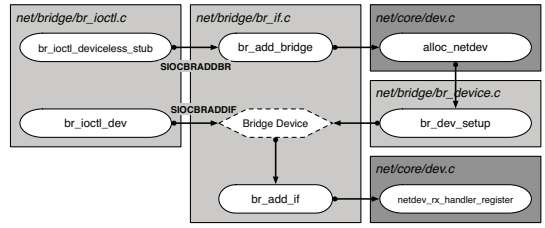

Figure 2: Linux bridge configuration; adding a bridge and a bridge port

The Linux bridging module has two separate configuration interfaces exposed to the user-space of the operating system. The first, ioctl interface offers an interface that can be used to create and destroy bridges in the operating system, and to add and remove existing network interfaces to/from the bridge. The second, sysfs based interface allows the management of bridge and bridge port specific parameters. Figure 2 presents a high level overview of the kernel ioctl process, that creates and initializes the bridge object, and adds network interfaces to it. The functions on dark grey areas are in the generic kernel, while the lighter areas are in the bridge.

The creation of a new bridge begins with the ioctl command *SIOCBRADDBR* that takes the bridge interface name as a parameter. The ioctl command is handled by the `br_ioctl_deviceless_stub` function, as there is no bridge device to attach the ioctl handler internally. The addition of a new bridge calls the function `br_add_bridge`, that creates the required bridge objects in the kernel, and eventually calls the `alloc_netdev` function to create a new netdevice for the bridge. The allocated netdevice is then initialized by the `br_dev_setup` call, including assigning the bridge device specific ioctl handler `br_dev_ioctl` to the newly allocated netdevice. All subsequent bridge specific ioctl calls are done on the newly created bridge device object in the kernel.

Ports are added to bridges by the ioctl command *SIOCBRADDIF*. The ioctl command takes the bridge device and the index of the interface to add to the bridge as parameters. The ioctl calls the bridge device ioctl handler (`br_dev_ioctl`), that in turn calls the `br_add_if` function. The function is responsible for creating and setting up a new bridge port by allocating a new `net_bridge_port` object. The object initialization process automatically sets the interface to receive all traffic, adds the network interface address for the bridge port to the forwarding database as a local entry, and attaches the interface as a slave to the bridge device. Finally, the function also calls the `netdev_rx_handler_register` function that sets the `rx_handler` of the network interface to `br_handle_frame`, that enables the interface to start processing incoming frames as a part of the bridge.

2.3 Frame processing

The Linux bridge processing flow begins from lower layers. As mentioned above, each network interface that acts as a bridge interface, will have a `rx_handler` set to `br_handle_frame`, that acts as the entry point to the bridge frame processing code. Concretely, the `rx_handler` is called by

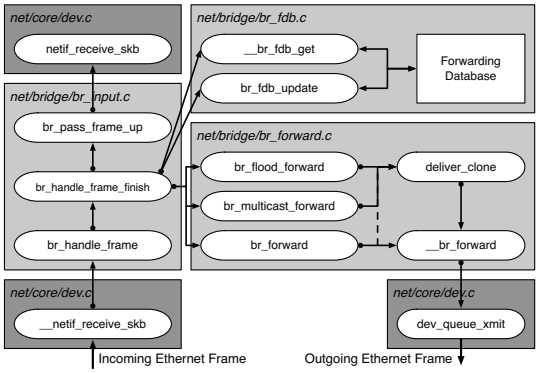

Figure 3: Architectural overview of the Linux bridge module I/O

the device-independent network interface code, in `__netif_receive_skb`. Figure 3 presents the processing flow of an incoming frame, as it passes through the Linux bridge module to a destination network interface queue.

The `br_handle_frame` function does the initial processing on the incoming frame. This includes doing initial validity checks on the frame, and separating control frames from normal traffic, because typically these frames are not forwarded in local area networks. The bridge considers any frame that has a destination address prefix of `01:80:C2:00:00` to be a control frame, that may need specialized processing. The last byte of the destination MAC address defines the behavior of the link local processing. Currently, Ethernet pause frames are automatically dropped, STP frames are either passed to the upper layers if it is enabled on the bridge, or forwarded, when it is disabled. Finally, if a forwarding decision is made, and the bridge is in either forwarding or learning mode, the frame is passed to `br_handle_frame_finish`, where the actual forwarding processing begins.

The `br_handle_frame_finish` function first updates the forwarding database of the bridge with the source MAC address, and the source interface of the frame by calling `br_fdb_update` function. The update either inserts a new entry into the forwarding database, or updates an existing entry.

Next, the processing behavior is decided based on the destination MAC address in the Ethernet frame. Unicast frames will have the forwarding database indexed with the destination address by using the `__br_fdb_get` function to find out the destination `net_bridge_port` where the frame will be forwarded to. If a `net_bridge_fdb_entry` object is found, the frame will be directly forwarded through the destination interface by the `br_forward` function. If no entry is found for the unicast destination Ethernet address, or the destination address is broadcast, the processing will call the `br_flood_forward` function. Finally, if the frame is a multi-destination frame, the multicast forwarding database is indexed with the complete frame. If selective multicasting is used and a multicast forwarding entry is found from the database, the frame is forwarded to the set of bridge ports for that multicast address group by calling the `br_multicast_forward` function. If no entry is found or selective multicasting is disabled, the frame will be handled as a broadcast Ethernet frame and forwarded by the `br_flood_forward` function.

In cases where the destination MAC address of the incoming frame is multi- or broadcast, the bridge device is set to receive all traffic, or the address is matches one of the local interfaces, a clone of the frame is also delivered upwards in the local network stack by calling the `br_pass_frame_up` function. The function updates the bridge device statistics, and passes the incoming frame up the network stack by calling the device independent `netif_receive_skb` function, ending the bridge specific processing for the frame.

The forwarding logic of the Linux bridge module is implemented in three functions: `br_forward`, `br_multicast_forward`, and `br_flood_forward`, to forward unicast, multicast, and broadcast or unknown unicast destination Ethernet frames, respectively. The simplest of the three, the `br_forward` function checks whether the destination bridge interface is in forwarding state, and then either forwards the incoming frame as is, clones the frame and forwards the cloned copy instead by calling the `deliver_clone` function, or doing nothing if the bridge interface is blocked. The `br_multicast_forward` function performs selective forwarding of the incoming Ethernet frame out of all of the bridge interfaces that have registered multicast members for the destination multicast address in the Ethernet frame, or on interfaces that have multicast routers behind them. The `br_flood_forward` function iterates over all of the interfaces in the bridge, and delivers a clone of the frame through all of them except the originating interface. Finally, all three types of forwarding functions end up calling the `__br_forward` function that actually transfers the frame to the lower layers by calling the `dev_queue_xmit` function of the interface.

3. TECHNICAL DETAILS

The Linux bridge module has two specific components that are explored in detail in this section. First, the functionality of the forwarding database is described in detail. Secondly, an overview of the IGMP snooping and selective multicasting subsystem of the Linux bridge is given, concentrating on the functional parts of the design.

3.1 Forwarding database

The forwarding database is responsible for storing the location information of hosts in the LAN. Figure 4 shows the indexing mechanism for the forwarding table, and the structure of the forwarding database array. Internally, the forwarding database is an array of 256 elements, where each element is a singly linked list holding the forwarding table entries for the hash value. The hash value for all destination MAC addresses is calculated by the `br_hash_mac` function.

The hashing process begins by extracting the last four bytes of the MAC address, creating a 32 bit identifier. The last four bytes are chosen because of the address organization in MAC addresses. Each 48 bit address consists of two parts. The first 24 bits specify an Organizationally Unique Identifier (OUI) that is assigned to the organization that issued the address. The last 24 bits specify an identifier that is unique within the OUI. The fragment of the MAC value

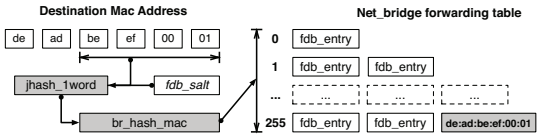

Figure 4: Linux bridge forwarding table indexing

used by the bridge contains a single byte of the OUI and all three bytes of the OUI specific identifier. This guarantees a sufficiently unique identifier, while still allowing efficient hashing algorithms to be used.

The MAC address fragment, along with a randomly generated `fdb_salt` value is passed to a generic single word hashing function in the Linux kernel, called `jhash_1word`. The resulting 32 bit hash value is then bounded to the maximum index in the hash array (i.e., 255) to avoid overflowing. The forwarding table entry for the destination MAC address is found by iterating over the linked list of the hash array element, pointed by the truncated hash value.

Unused entries in the forwarding table are cleaned up periodically by the `br_cleanup` function, that is invoked by the garbage collection timer. The cleanup operation iterates over all the forwarding database entries and releases expired entries back to the forwarding table entry cache. During iteration, the function also keeps track of the next invocation time of the cleanup operation. This is done by keeping track of the next expiration event after the cleanup invocation, based on the expiration times of the forwarding table entries that are still active during the cleanup operation.

3.2 IGMP Snooping

The IGMP snooping features of the Linux kernel bridge module allows the bridge to keep track of registered multicast entities in the local area network. The multicast group information is used to selectively forward incoming multicast Ethernet frames on bridge ports, instead of treating multicast traffic the same way as broadcast traffic. While IGMP is a network layer protocol, the IPv4 multicast addresses directly map to Ethernet addresses on the link layer. Concretely, the mapping allows local area networks to forward IPv4 multicast traffic only on links that contain hosts that use it. This can have a significant effect in the traffic characteristics of the local area network, if multicast streaming services, such as IPTV are used by several hosts.

IGMP snooping functionality consists of two parts in the Linux kernel: First, multicast group information is managed by receiving IGMP messages from end hosts and multicast capable routers on bridge ports. Next, based on the multicast group information, the forwarding decision of the bridge module selectively forwards received multicast frames on the ports that have reported a member on the multicast group address in the Ethernet frame destination address field. This paper discusses the latter part of the operation by going over the details of the multicast forwarding database, and the multicast forwarding database lookup.

Figure 5 presents an overview of the multicast forwarding

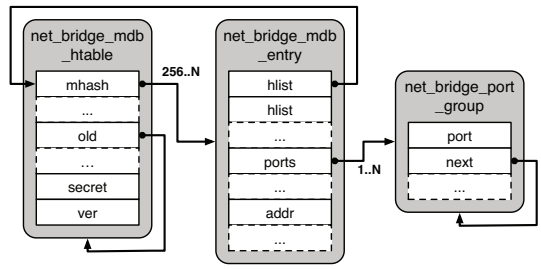

Figure 5: Linux bridge multicast forwarding database structure

database structure, and the relationships between the main data structures. The multicast forwarding database is contained in the `net_bridge_mdb_htable` data structure. The field `mhash` points to a hash array of linked list objects, similar to the normal forwarding database. The significant difference between the normal forwarding database and the multicast forwarding database is that the hash table is dynamically resized, based on the number of multicast groups registered by the operating system, either from local or remote sources. To support the efficient resizing of the database, a special field `old` is included in the data structure. This field holds the previous version of the multicast forwarding database. The previous version is temporarily stored because the rehashing operation of the multicast forwarding database is done in parallel with read access to the previous database. This way, the rehashing operation does not require exclusive access to the multicast forwarding database, and the performance of the multicast forwarding operation does not significantly degrade. After the rehash operation is complete, the old database is removed. Finally, the data structure also contains the field `secret`, that holds a randomly generated number used by the multicast group address hashing to generate a hash value for the group.

Each multicast group is contained in a `net_bridge_mdb_entry` data structure. The data structure begins with a two element array `hlist`. These two elements correspond to the position of the multicast group entry in the two different versions of the multicast forwarding database. The current version of the multicast forwarding table is defined by the `net_bridge_mdb_htable->ver` field, that will be either 0 or 1. The `ports` field contains a pointer to a `net_bridge_port_group` data structure that contains information about a bridge port that is a part of this multicast group. Finally, the `addr` field contains the address of the multicast group.

The third primary data structure for the multicast forwarding system is the `net_bridge_port_group`. The data structure holds a pointer to the bridge *port*, and a pointer to the next `net_bridge_port_group` object for a given `net_bridge_mdb_entry` object. The data structure also contains the multicast group address and various timers related to the bookkeeping of the multicast group information.

The multicast forwarding database lookup is similar to the forwarding table lookup. Figure 6 presents an overview of the operation. The hashing function takes two separate

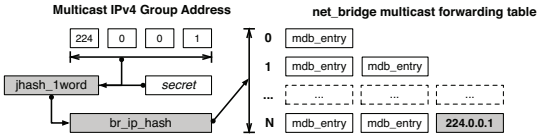

Figure 6: Linux bridge multicast forwarding database indexing

values and passes them to a generic hashing function in the Linux kernel (e.g., `jhash_1word`), similar to the MAC address hashing operation. For IPv4, the full multicast group address and the contents of the field `net_bridge_mdb_htable->secret` field are passed to the hashing function, resulting in a hash value. IPv6 uses a different hashing function that takes the full 128-bit address as an array of 4 32-bit integers. The hash value is then bounded to the maximum index of the multicast forwarding database hash array. As with the normal forwarding table, the correct `net_bridge_mdb_entry` is found by iterating over all the elements in the linked list, pointed by the bounded hash value.

4. EXPERIMENTATION

Packet processing on generic hardware is generally seen as memory intensive work [3, 4]. The experimentation in this paper explore the processing distribution between the different components of the system during the forwarding process.

4.1 Evaluation Setup

Figure 7 presents the experiment environment. It consists of a Spirent Testcenter traffic generator, and a Linux server using kernel version 3.5.3, with a bridge acting as the Device Under Test (DUT). The Spirent Testcenter generates a full duplex stream of Ethernet frames that are forwarded by the DUT using two 1Gbps network interface ports. The Linux kernel on the server collects performance statistics during the tests using the built-in profiling framework in the kernel.

The performance framework is controlled from the user space by the *perf* tool [13]. The tool offers commands to manage the performance event data collection, and to study the results. To collect performance event data, the user defines a list of either pre-defined performance events that are mapped to CPU-specific performance events by the tool, or raw performance events that can typically be found from the reference guide of the CPU or architecture model.

To generate usable performance event data, the Spirent Testcenter was used to run the RFC 2889 [9] forwarding test with 64 byte frames to determine the maximum forwarding rate of the DUT. The forwarding test performs several forwarding runs, and determines the maximum forwarding rate of the DUT by using a binary search like algorithm to narrow the forwarding rate to within a percent of the maximum. Then, five separate tests using the maximum forwarding rate with performance event data collection were run with one and 1024 Ethernet hosts on each port. The reason the performance event data collection was done this way was to eliminate the effects of frame discarding from the results, due to receiving too many frames from the traffic generator.

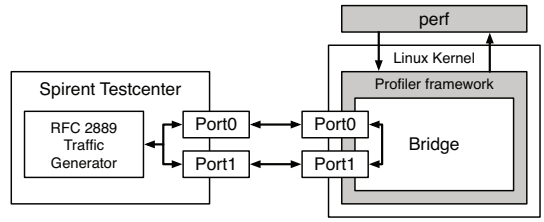

Figure 7: Experiment environment

The kernel was instrumented to collect two different performance events during the testing: used clock cycles, and cache references and misses. The cycles can be used as an estimator on the distribution of CPU processing inside the kernel. Cache references and cache misses can be used to estimate the workload of the memory subsystem in two ways. Each cache reference and miss can be likened to an operation in the CPU, that requires the program to access the main memory of the system. Cache reference occurs, when the accessed information is found in a cache, avoiding an expensive main memory access. Conversely, a cache miss happens when the information is not available in any of the caches of the CPU, and the operation requires an expensive access to the main memory of the computer.

4.2 Results

Table 1 presents the distribution of work between the different subsystems of the Linux kernel during the forwarding test with 64 byte frames. The results are given as a percent of the total number of event counters collected in the tests. The work is divided into four different subsystems: the bridge module, the network interface card driver and the network device API, the locking mechanism of the kernel for shared data structures, and the memory management.

Table 1: Performance event data distribution for RFC 2889 forwarding test

	Hosts					
	2	2048	2	2048	2	2048
Subsystem	**Cycles%**		**Cache Ref%**		**Cache Miss%**	
Interface	45.7%	40.5%	55.0%	42.2%	77.5%	77.9%
Bridge	21.0%	29.2%	11.1%	31.5%	4.2%	3.8%
Memory/IO	19.6%	17.2%	28.8%	22.0%	5.1%	5.4%
Locks	13.7%	13.2%	5.2%	4.3%	13.2%	12.9%

Almost 46% of the CPU cycles are spent in the device driver and the network device independent layer of the network stack. Next, the Linux bridging module and the memory management of the kernel are spending roughly 20% of the cycles each. Finally, the locking mechanism of the kernel is taking up the last 15% of cycles. As the number of hosts in the test increases from two to 2048, we can see that the bridge uses a larger portion of the overall cycles. The increase in used cycles is related to the organization of the hash array in the forwarding database.

The network interface and the device driver are also responsible for 55% of the cache references, and 78% of the cache misses, when there are two hosts in the LAN. We can also see a similar trend with the Linux bridging module here, as with

the cycle use. When the number of hosts increases from two to 2048, the Linux bridging module uses significantly larger portion of memory operations (and thus, caching operations) to update and query the forwarding database.

Table 2 presents the distribution of work in the Linux bridge module between the four busiest functions during the forwarding test. The results are given as a percent of the total number of event counters collected in the tests. Note that the table only holds four of the 13 different functions that participate in the DUT forwarding operation.

Table 2: Performance event distribution for RFC2889 forwarding test in the bridge module

	Hosts					
	2	2048	2	2048	2	2048
Function	**Cycles%**		**Cache Ref%**		**Cache Miss%**	
nf_iterate	19.6%	13.2%	2.3%	3.4%	12.1%	8.8%
br_fdb_update	18.2%	26.1%	42.0%	39.0%	0.1%	0.3%
br_handle_frame	13.5%	8.6%	2.7%	1.1%	3.7%	6.9%
__br_fdb_get	10.0%	23.6%	41.3%	42.9%	0.1%	0.6%

The most interesting piece of information can be seen in these results. During testing, most cycles in the Linux bridging module are not used by a bridge-specific function. The `nf_iterate` function is used by the *netfilter* module to iterate over the rules that have been specified in the system. All of the work performed by the `nf_iterate` function during the frame forwarding tests is essentially wasted, as the system had no netfilter rules defined nor does the bridging module require netfilter for any operational behavior.

We can also see from the table that most of the memory related operations are performed by the two forwarding database functions `br_fdb_update` and `__br_fdb_get`. When the number of hosts during testing is increased to 2048, the two functions also consume most of the cycles during testing. The reason for the increased processor cycle usage with increased number of hosts is explained by the architecture of the forwarding database. As mentioned in 3.1, the forwarding database consists of an array of 256 elements, where each element is a linked list. The hashing function assigns the forwarding database entry for the MAC address to one of the linked lists. Thus, the more hosts the system sees, the longer the average length of the chain for a single linked list will become. The entries in the linked lists are in arbitrary order, which requires a linear seek through the full list. This significantly increases the number of clock cycles required to find the MAC address from the linked list.

As can be seen from the table, the number of cache references stays roughly the same while the number of hosts is increased. In addition, the forwarding database in both cases fits into the system cache, as the number of misses during the forwarding database functions is insignificant. The majority of cache misses occur in the various netfilter related functions of the overall frame processing.

5. CONCLUSION

Ethernet based LANs are the building block of IP based networks, and the network application ecosystem. Local area networks are built by bridges that connect multiple Ethernet links into a single larger Ethernet cloud.

The Linux kernel contains a bridge module that can be used to create local area networks by combining network interface ports of a computer under a single bridge. While Linux bridges are not able to compete with specialized vendor hardware in performance, Linux bridging can be used in environments where performance is not the priority.

The experimentation conducted for this paper explored the performance characteristics of the Linux kernel during the bridge operation. The results show that most of the processing time is consumed by the device driver and the network interface, instead of the bridge. We can also see that modern most of the packet forwarding to occur inside the caches of the CPU. The evaluation also shows a significant increase in processing requirements in the bridge module, when the number of hosts in the LAN is significant increased.

6. REFERENCES

[1] D. Allan et al. Shortest path bridging: Efficient control of larger ethernet networks. *IEEE Communications Magazine*, 48:128–135, Oct. 2010.

[2] G. Chiruvolu, A. Ge, D. Elie-Dit-Cosaque, M. Ali, and J. Rouyer. Issues and approaches on extending Ethernet beyond LANs. *Communications Magazine, IEEE*, 42(3):80 – 86, March 2004.

[3] N. Egi et al. Towards high performance virtual routers on commodity hardware. CoNEXT '08. ACM.

[4] N. Egi et al. Forwarding path architectures for multicore software routers. PRESTO '10. ACM, 2010.

[5] W. Fenner. Internet Group Management Protocol, Version 2. RFC 2236, Internet Engineering Task Force, November 1997.

[6] A. Greenberg et al. VL2: a scalable and flexible data center network. In *SIGCOMM*. ACM, 2009.

[7] C. Guo et al. Dcell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, pages 75–86. ACM, 2008.

[8] C. Guo et al. BCube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM*. ACM, 2009.

[9] R. Mandeville and J. Perser. Benchmarking Methodology for LAN Switches. RFC 2889, Internet Engineering Task Force, August 2000.

[10] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. Mogul. SPAIN: COTS data-center ethernet for multipathing over arbitrary topologies. In *NSDI*. USENIX, 2010.

[11] R. Niranjan Mysore et al. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, pages 39–50. ACM, 2009.

[12] Opensolaris rbridge (IETF TRILL) support. `http://hub.opensolaris.org/bin/view/Project+rbridges/`.

[13] perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org`.

[14] R. J. Perlman. Rbridges: Transparent routing. In *INFOCOM*, pages 1211–1218, 2004.

[15] Media Access Control (MAC) Bridges. Standard 802.1D, IEEE, 2004.

[16] Virtual Bridged Local Area Networks. Standard 802.1Q-2005, IEEE Computer Society, 2005.

Pasi Sarolahti (ed.) Proceedings of Seminar on Network Protocols in Operating Systems

Aalto-ST 1/2013

ISBN 978-952-60-4997-7 (pdf)
ISSN-L 1799-4896
ISSN 1799-4896
ISSN 1799-490X (pdf)

Aalto University
School of Electrical Engineering
Department of Communications and Networking
www.aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL DISSERTATIONS

Aalto University